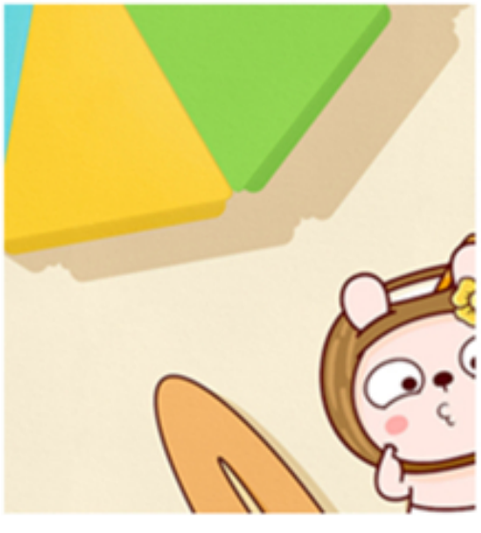
最新IT书籍PDF
最新IT书籍PDF
最新IT书籍PDF
最新IT书籍PDF



版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

仅供学术交流使用

HZ BOOKS 华章IT

[PACKT] PUBLISHING

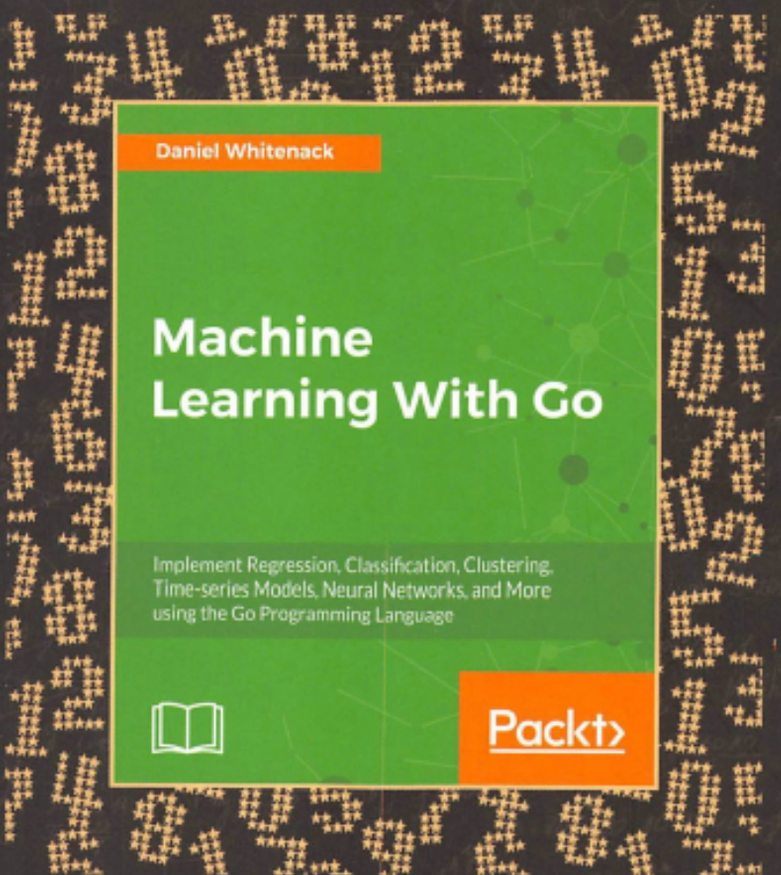
数据科学与工程技术丛书

机器学习

Go语言实现

[美] 丹尼尔·怀特纳克（Daniel Whitenack） 著

谢文江 姜明魁 译



MACHINE LEARNING WITH GO

机械工业出版社
China Machine Press
仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

内容简介

本书由资深数据科学家撰写，通过大量Go语言实例，深入浅出地介绍机器学习理论。从基础的数据收集和整理，到常见机器学习技术介绍，再到最后的神经网络和深度学习，所有这些内容均有详细的Go语言实现。

全书共9章，第1章详细讲解如何从各种数据来源中收集、组织和解析实际工作数据；第2章介绍如何在Go程序中生成矩阵，以及如何执行各种类型的矩阵运算、可靠的汇总数据分析、描述和可视化分布、量化假设、转换数据集；第3章讨论评估和验证；第4~6章介绍如何使用Go软件包实现基本的机器学习技术（回归、分类、集群等）；第8章介绍利用神经网络进行回归、分类和图像处理的技术；第9章讨论如何从笔记本电脑中获取这些模型和应用程序，并在数据流水线中以生产规模运行它们。此外，附录部分还介绍了与机器学习工作流程相关的算法、优化方法和技术。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

华章IT
HZBOOKS | Information Technology

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

数据科学与工程技术丛书

MACHINE LEARNING WITH GO

机器学习

Go语言实现

[美] 丹尼尔·怀特纳克（Daniel Whitenack） 著
谢文江 姜明魁 译

机械工业出版社
China Machine Press

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

图书在版编目（CIP）数据

机器学习：Go 语言实现 /（美）丹尼尔·怀特纳克（Daniel Whitenack）著；谢文江，姜明魁译 . —北京：机械工业出版社，2018.9
（数据科学与工程技术丛书）
书名原文：Machine Learning With Go

ISBN 978-7-111-60979-7

I. 机… II. ①丹… ②谢… ③姜… III. 程序语言 - 程序设计 IV. TP312

中国版本图书馆 CIP 数据核字（2018）第 218894 号

本书版权登记号：图字 01-2017-7516

Daniel Whitenack: *Machine Learning With Go* (ISBN: 978-1-78588-210-4).

Copyright © 2017 Packt Publishing. First published in the English language under the title "Machine Learning With Go".

All rights reserved.

Chinese simplified language edition published by China Machine Press.

Copyright © 2018 by China Machine Press.

本书中文简体字版由 Packt Publishing 授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

本书不仅深入浅出地介绍了利用 Go 语言构建预测模型的技术，还会帮助读者理解如何在真实场景中应用机器学习工作流程。第 1 章～第 3 章讲述了在机器学习流程中如何准备和分析数据；第 4 章～第 7 章详细介绍了回归、分类、集群、时间序列和异常检测等技术；第 8 章和第 9 章对神经网络、深度学习、部署、分布分析和模型进行了深入探究；附录介绍了与机器学习相关的算法 / 技术。本书适合作为对 Go 感兴趣的数据科学家、分析师、工程师和相关专业学生的参考书。

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：冯秀泳　　责任校对：殷　虹
印　　刷：北京市兆成印刷有限责任公司　　版　　次：2018 年 10 月第 1 版第 1 次印刷
开　　本：185mm×260mm　1/16　　印　　张：14.25
书　　号：ISBN 978-7-111-60979-7　　定　　价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：(010) 88378991　88361066　　投稿热线：(010) 88379604
购书热线：(010) 68326294　88379649　68995259　　读者信箱：hzjsj@hzbook.com

版权所有·侵权必究
封底无防伪标均为盗版
本书法律顾问：北京大成律师事务所　韩光 / 邹晓东

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

译　者　序

机器学习是人工智能的核心，是以机器学习为手段解决人工智能中的问题。它在最近 30 多年已经发展为一门多领域的交叉学科，涉及概率论、统计学、计算复杂性理论等多门学科。其内容主要是设计和分析一些可以让计算机自动学习的算法，用以从繁杂的数据中自动进行分析来获得规律，并利用该规律对未知数据进行预测。

当前市面上关于机器学习的书籍举不胜举，其中大部分都是使用 Python 或者 R。本书独辟蹊径地使用 Go 语言来进行机器学习，不仅克服了其他语言中容易忽视的一些问题（比如说维持高级别的数据完整性），还会利用 Go 的独特功能，帮助读者构建优良、高效且易维护的机器学习代码。

本书不仅深入浅出地介绍了机器学习的理论，还使用了大量的 Go 语言实例。从基础的数据收集和整理，到常见机器学习技术介绍，再到最后的神经网络和深度学习，所有这些内容均有详细的 Go 语言实现。

不仅如此，本书所使用的软件包、部署工具以及工作流程在业内已经广泛使用，读者可以轻松地将其整合到自己的机器学习和数据分析流程中。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

关于作者

Daniel Whitenack（@dwhitena）博士是一位工作在 Pachyderm（@pachydermIO）的资深数据科学家，开发出了革命性的、分布式的数据流水线，其中包括预测模型、数据可视化、统计分析等，在世界各地的会议（GopherCon、JuliaCon、PyCon、ODSC、Spark Summit 等）上做过演讲。他目前在普渡大学（@LifeAtPurdue）教授数据科学 / 工程，同时与 Ardan 实验室（@ardanlabs）共同维护着 Jupyter 项目的 Go 内核，并积极为各种开源数据科学项目做出贡献。

在这里我想感谢我的妻子，感谢她在我撰写此书时给予的无限耐心和支持。

同时我还想感谢许多热心人和数据科学家对我的指导、合作以及鼓励。他们是 Bill Kennedy、Brendan Tracey、Sebastien Binet、Alex Sanchez、Pachyderm 的整个团队（包括 Joey Zwicker 和 Joe Doliner）、Chris Tava、Mat Ryer、David Hernandez、Xuanyi Chew、Minio 的团队（包括 Anand Babu Periasamy 和 Garima Kapoor），以及其他许多帮助过我的人。

Soli Deo Gloria

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

关于审校者

Niclas Jern，自从 Go 1.0 起就一直使用 Go 语言解决大量有趣的问题。他在埃博学术大学获得计算机工程的硕士学位，专业是软件工程。

他喜爱用 Go 以及其他编程语言去解决问题，尤其是数据处理和机器学习领域的问题。他的爱好是散步、在健身房举铁健身，偶尔在 http://www.njern.co 发言，以及同家人享受天伦之乐。

Niclas 目前就职于 Walkbase，这是他与埃博学术大学的同学联合创办的一家公司。在公司中他主要领导工程团队，解决来自革命性的零售分析中的数据处理问题。

Richard Townsend，还在华威大学攻读学士学位的时候，就成了 2014 年 GoLearn 的最高贡献者（这也是 GoLearn 有许多奇怪行为的原因）。自那时起，他便在英国顶级科技公司工作，工作范围包括从嵌入式系统到安卓操作系统框架的所有内容，目前正致力于网络浏览器的优化工作。他花费了大量的时间从事情感分析工作（在情感分析方面合著了两篇论文），以及其他自然语言处理任务，例如使用最新的深度学习技术进

行词性标注。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

前　言

在位于科技前沿的公司以及越来越多的大型企业中，机器学习和人工智能似乎都风靡一时。数据科学家正在尝试使用机器学习来完成任何事情，从驾驶汽车到画猫。但是，如果关注数据科学界的动态，很可能会看到类似于 Python 和 R 用户之间的**语言战争**。这些语言在机器学习交流中占据主导地位，而且往往被当成将机器学习集成到组织中的唯一选择。本书中将探索第 3 个选项：由 Google 创造的开源编程语言，Go。

Go 的独特功能以及 Go 程序员的思维方式，可以帮助数据科学家克服他们遇到的一些常见困难。尤其是数据科学家（很不幸地）被公认为会产生糟糕、低效和不易维护的代码。本书将解决这个问题，并将清楚地阐明，如何在机器学习方面富有成效，同时还能生产出保持高级别完整性的应用程序。书中还会帮助读者应对在现有工程学组织中有关完整性分析和机器学习代码的常见挑战。

本书将使读者成为富有成效的、有创新精神的数据分析师，帮助读者利用 Go 来构建强大而有价值的应用程序。为此，本书会清楚地介绍在 Go 世界中机器学习的技术和编程方面的内容，同时也将指导读者理解现实分析工作中合理的工作流程和理念。

本书内容

在机器学习流程中准备和分析数据：

- 第 1 章，*数据的收集和组织*，包含了收集、组织和解析数据，这些数据来自本地和远端，或者需要保存到本地和远端。阅读完本章，读者会理解如何与存储在多个位置、不同格式的数据做交互，如何解析和清洗数据，如何输出清洗过、解析过的数据。
- 第 2 章，*矩阵、概率论和统计学*，涵盖了将数据组织成矩阵和矩阵运算。阅读完本章，读者将了解如何在 Go 程序中生成矩阵，以及如何利用这些矩阵执行各种类型的矩阵运算。本章会介绍统计度量和日常数据分析工作的操作关键。

读者读完本章之后，还将了解如何执行可靠的汇总数据分析、描述和可视化分布、量化假设以及转换数据集（例如降维）。

- 第 3 章，*评估和验证*，本章主要介绍评估和验证，评估和验证是衡量机器应用性能和确保其通用化的关键。一旦读者完成本章，将会了解评估模型性能的各种度量指标，以及各种用以更通用地验证模型的技术。

机器学习的技术：

- 第 4 章，*回归*，本章解释了回归是一种广泛使用的连续变量建模技术，同时也是其他模型的基础。回归生成能立刻解释的模型。因此当开始在机构中引入预测功能时，回归可以提供一个很好的起点。
- 第 5 章，*分类*，分类作为一种与回归不同的机器学习技术，其目标变量通常是分类的或标记的。例如，分类模型可以将电子邮件分类为垃圾邮件或者非垃圾邮件，或者将网络数据分类为欺诈性或非欺诈性的数据等。
- 第 6 章，*集群*，集群是一种用于形成样本分组的无监督机器学习技术。在本章的最后，读者将能够自动形成数据点的分组，用以更好地理解其结构。
- 第 7 章，*时间序列和异常检测*，本章介绍了用于对时间序列数据进行建模的技术，如股票价格、用户事件等。阅读本章后，读者将会理解如何评估时间序列中的各种术语，建立时间序列模型，并检测时间序列中的异常情况。

机器学习探究：

- 第 8 章，*神经网络和深度学习*，介绍了利用神经网络进行回归、分类和图像处理的技术。阅读本章之后，读者将了解如何以及何时应用这些更复杂的建模技术。
- 第 9 章，*部署、分布分析和模型*，使读者能够将整个课程中开发的模型应用部署到生产环境，并对生产规模数据进行分布处理。本章说明了如何利用本书使用的现有代码轻松完成这两件事情，而不用对代码进行重大修改。
- 附录，*与机器学习相关的算法 / 技术*，会在本书全书中被引用，并且将提供与机器学习工作流程相关的算法、优化和技术的信息。

阅读本书需要的准备

为了运行本书中的示例并尝试使用本书中涵盖的技术，通常需要以下内容：

- 可以访问一个类似 bash 的 shell。
- 一个完整的 Go 环境，包括 Go、一个编辑器以及相关的默认或定制的环境变量。例如，可以按照在 https://www.goinggo.net/2016/05/installing-go-and-your-workspace.html 上的指南进行操作。
- 各种 Go 的依赖包。这些可以通过命令 `go get` 得到。

然后，为了运行与一些高级的机器学习主题（如数据流水线和深度学习）相关的示例，需要准备一些额外的内容：

- 需要安装或部署 Pachyderm。可以按照链接中的文档来获取 Pachyderm，并在本地或在云上运行：http://pachyderm.readthedocs.io/en/latest/。
- 一个可运行的 Docker（https://www.docker.com/community-edition#/download）。
- 安装一个 TensorFlow。要在本地安装 TensorFlow，可以在 https://www.tensorflow.org/install/ 上阅读安装指南。

谁适合读本书

本书对于以下类型的读者会有所帮助：

1. 对机器学习和数据分析感兴趣的 Go 程序员
2. 对 Go 感兴趣的数据科学家 / 分析师 / 工程师，并希望将其整合到他们的机器学习和数据分析工作流程中

约定

许多 Go 代码片段不包括 `package main` 和 `func main(){}` 主函数。除非特别说明，Go 代码片段都是在一个必要的基础架构中编译。在本书中将假定代码示例被编译在一个叫做 `myprogram` 的目录中，因此会被编译为一个名为 `myprogram` 的二进制文件。然而，读者将认识到，代码可以被复制到 `GOPATH/src` 目录下的任何文件夹中，或者被编译成根据读者的喜好命名的二进制文件。

在本书中，为了区分不同类型的信息，采用了不同的文本样式。下面是这些样式的一些例子以及它们的含义说明。Go 语言中的文本、数据库表名、文件夹名、文件名、文件扩展名、路径名、虚拟 URL、用户输入和 Twitter 句柄中的代码如下段所示："接下来的代码行读取链接并将其分配给 `BeautifulSoup` 功能。"代码块如下所示：

```
#import packages into the project
from bs4 import BeautifulSoup
from urllib.request import urlopen
import pandas as pd
```

当需要提醒读者注意代码块的特定部分时，相关行或项目以粗体显示：


```
 [default] exten => s,1,Dial(Zap/1|30)
 exten => s,2,Voicemail(u100)
 exten => s,102,Voicemail(b100)
exten => i,1,Voicemail(s0)
```

命令行输入或输出如下所示：

```
C:\Python34\Scripts> pip install -upgrade pip
C:\Python34\Scripts> pip install pandas
```

新术语和**重要词汇**以粗体显示。在屏幕上看到的单词，例如菜单或对话框，在本书中会如下所示："为了下载新模块，我们将转到：**文件** | **设置** | **项目名称** | **项目解释器**"。

警告或重要的注意事项使用这样的图标。

提示和技巧使用这样的图标。

下载示例代码及彩色图像

本书的示例源码及所有截图和样图，可以从 http://www.packtpub.com 通过个人账号下载，也可以访问华章图书官网 http://www.hzbook.com，通过注册并登录个人账号下载。

目　录

译者序

前言

第 1 章　数据的收集和组织 ……1

1.1　数据处理-Gopher 方式……2

1.2　Go 语言收集和组织数据的最佳实践……4

1.3　CSV 文件……5

1.3.1　从文件中读取 CSV 数据……5

1.3.2　处理非预期的域……6

1.3.3　处理非预期的类型……7

1.3.4　用数据帧操作 CSV 数据……9

1.4　JSON……11

1.4.1　JSON 的解析……11

1.4.2　JSON 的输出……14

1.5　SQL-like 数据库……14

1.5.1　连接到一个 SQL 数据库……15

1.5.2　查询数据库……15

1.5.3　修改数据库……17

1.6　缓存……17

1.6.1　在内存中缓存数据……17

1.6.2　在本地磁盘中缓存数据……18

1.7　数据版本控制……19

1.7.1　Pachyderm 术语……20

1.7.2　部署 / 安装 Pachyderm……20

1.7.3　创建用于数据版本控制的数据仓库……21

1.7.4　把数据存储到数据仓库中…21

1.7.5　从版本化的数据仓库中获取数据……22

1.8　参考书目……22

1.9　小结……23

第 2 章　矩阵、概率论和统计学……24

2.1　矩阵和向量……24

2.1.1　向量……24

2.1.2　向量操作……25

2.1.3　矩阵……26

2.1.4　矩阵操作……27

2.2　统计学……29

2.2.1　分布……29

2.2.2　统计方法……30

2.2.3　分布可视化……34

2.3　概率论……39

2.3.1　随机变量……40

2.3.2　概率测量……40

2.3.3　独立和条件概率……40

2.3.4　假设检验……41

2.4　参考书目……43

2.5　小结……44

第 3 章　评估和验证……45
3.1　评估……45
3.1.1　连续指标……46
3.1.2　分类指标……49
3.2　验证……55
3.2.1　训练和测试集……56
3.2.2　保留集……59
3.2.3　交叉验证……60
3.3　参考书目……61
3.4　小结……62
第 4 章　回归……63
4.1　理解回归模型的术语……63
4.2　线性回归……64
4.2.1　线性回归概述……64
4.2.2　线性回归假设和陷阱……66
4.2.3　线性回归示例……66
4.3　多元线性回归……78
4.4　非线性和其他类型的回归……81
4.5　参考书目……85
4.6　小结……86
第 5 章　分类……87
5.1　理解分类模型的术语……87
5.2　逻辑回归……88
5.2.1　逻辑回归概述……88
5.2.2　逻辑回归的假设和陷阱……91
5.2.3　逻辑回归示例……92
5.3　k- 最近邻……103
5.3.1　kNN 概述……103
5.3.2　kNN 假设和陷阱……104
5.3.3　kNN 示例……105
5.4　决策树和随机森林……106
5.4.1　决策树和随机森林概述……107
5.4.2　决策树和随机森林的假设及陷阱……107
5.4.3　决策树示例……108
5.4.4　随机森林的例子……109
5.5　朴素贝叶斯……109
5.5.1　朴素贝叶斯概念及其重要假设……110
5.5.2　朴素贝叶斯例子……110
5.6　参考书目……111
5.7　小结……112
第 6 章　集群……113
6.1　理解集群模型术语……113
6.2　距离或相似度的度量……114
6.3　集群技术的评估……115
6.3.1　内部集群评估……115
6.3.2　外部集群评估……120
6.4　k-均值集群……120
6.4.1　k-均值集群综述……120
6.4.2　k-均值的假设和陷阱……122
6.4.3　k-均值集群的例子……123
6.5　其他集群技术……129
6.6　参考书目……130
6.7　小结……130
第 7 章　时间序列和异常检测……131
7.1　在 Go 中表示时序数据……131
7.2　理解时间序列的术语……134
7.3　与时间序列有关的统计……135
7.3.1　自相关……135
7.3.2　偏自相关……139
7.4　预测的自回归模型……141

7.4.1 自回归模型概述……141
7.4.2 自回归模型假设和陷阱…142
7.4.3 自回归模型示例……142
7.5 自回归移动平均和其他时间序列模型……151
7.6 异常检测……151
7.7 参考书目……153
7.8 小结……154

第 8 章 神经网络和深度学习……155

8.1 理解神经网络术语……155
8.2 构建一个简单的神经网络……157
8.2.1 网络中的节点……157
8.2.2 网络架构……158
8.2.3 为什么期望这种架构有作用……159
8.2.4 训练神经网络……160
8.3 使用简单的神经网络……165
8.3.1 在实际数据上训练神经网络……166
8.3.2 评估神经网络……168
8.4 引入深度学习……169
8.4.1 什么是深度学习模型……170
8.4.2 基于 Go 语言的深度学习…171
8.5 参考书目……177
8.6 小结……178

第 9 章 部署、分布分析和模型……179

9.1 在远程机器上可靠地运行模型…179
9.1.1 Docker 和 Docker 术语简介……180
9.1.2 Docker 化机器学习的应用……181
9.2 构建可拓展和可重现的机器学习流水线……191
9.2.1 搭建 Pachyderm 和 Kubernetes 集群……192
9.2.2 构建一个 Pachyderm 机器学习流水线……193
9.2.3 更新流水线并检查出处…202
9.2.4 缩放流水线阶段……204
9.3 参考书目……206
9.4 小结……206

附录 与机器学习相关的算法 / 技术……207

第 1 章
数据的收集和组织

调查结果显示，数据科学家至少有 90% 的时间是用于收集、组织和清洗数据，而不是用于训练或调试复杂的机器学习模型。为什么会这样呢？难道机器学习不才是最有趣的部分吗？为何会如此关注所收集数据的状态呢？首先，如果没有数据，机器学习模型将无法学习。虽然这是显而易见的事情，但需注意的是，构建模型的强大与否，有一部分是取决于所给予的数据。常言道，“输入是垃圾，产出也会是垃圾”。只有确定了收集的数据是相关并且干净的，机器学习的模型才能变得强大。也只有这样，才能如预想般操作数据，并产出有价值的结果。

当使用一种特定模型时，不是所有类型的数据都是合适的。例如，高维数据（例如文本数据）在一些特定模型中运行得就不是很好。另外一些模型假设变量是正态分布的，这种假设并非总是成立的。因此在收集需要的数据时应格外小心，并确保知道数据和模型是如何交互的。

数据科学家花如此多时间收集和组织数据还有另一个原因，那就是数据通常是凌乱的并且很难聚合。在大多数的组织中，数据经常以多种格式保存在多种系统中，并且拥有多种访问权限。通常情况下，很难像指定一个文件路径那样简单地为模型提供训练数据集合。

为了形成一个训练 / 测试集合，或者提供变量给一个模型以用于预测，可能需要处理各种各样的数据类型，例如 CSV、JSON、数据库中的表，等等。除此之外还可能需要转换其中一些数值。常见的转换包括解析时间格式、转化分类数据为数字数据、数值归一化、使用一些函数交叉处理数据。然而，不能总是假设一个特定变量的所有可能取值都会出现，或者能被同一种方式解析。

数据经常有数值缺失、类型混合、数值损坏的情况发生，如何处理这些场景将直接影响所构建模型的质量，因此需要仔细地收集、组织和理解所采用的数据。

虽然本书大部分内容的重点是各种各样的建模技术，但是读者仍应当将数据的收集、解析和组织当作是数据科学项目成功的一个（甚至是最）关键的组成部分。如果在机器学习的项目开发中，这部分数据没有保持高级别的完整性，那么从长远来看将会导致各种麻烦。

1.1 数据处理-Gopher 方式

和其他用于数据科学 / 分析的处理语言相比，Go 提供了强大的数据操作和解析能力。虽然其他语言（例如 Python 或者 R）能让用户很快地进行数据交互，但是这些语言经常倡导打破完整性的便利。如果在代码中经常采用动态和交互性数据探索，会导致行为变得奇怪。

以这个简单的 CSV 文件为例：

```
1,blah1
2,blah2
3,blah3
```

可以迅速写出一些 Python 代码去解析这个 CSV 文件，然后在无须了解数据是什么类型的情况下，输出整数列的最大值。

```
import pandas as pd

# Define column names.
cols = [
 'integercolumn',
 'stringcolumn'
 ]

# Read in the CSV with pandas.
data = pd.read_csv('myfile.csv', names=cols)

# Print out the maximum value in the integer column.
print(data['integercolumn'].max())
```

这个简单的程序将输出正确的结果：

```
$ python myprogram.py
3
```

如下所示，现在移除其中一个整数用于模拟数值丢失的情况。

```
1,blah1
2,blah2
,blah3
```

Python 程序随后就会打破数据结果的完整性，明确来讲，上述程序仍然会保持运

行，不会告诉用户有任何不一样的地方，仍然会产生数值，但却是一个不同类型的数值。

```
$ python myprogram.py
2.0
```

这种情况是无法接受的，一个整数数值可以在用户察觉不到任何变化的时候凭空消失。这种很难追踪的行为会对建模产生深远的影响。一般来说，一旦选择了动态类型和抽象的便利性，就需要接受这种行为上的变化。

重点并不是说 Python 无法处理这种行为，因为 Python 专家很快会认识到其实是有办法解决这个问题的。重点是这种便利性在默认情况下并没有提升完整性，因此很容易搬起石头砸自己的脚。

从另一方面来说，可以利用 Go 语言的静态类型和明确的错误处理方式，来确保数据的解析跟期望的一致。在这个简单例子中，可以毫不费力地写一些 Go 代码去解析的 CSV（现在暂不用关心具体的技术细节）：

```
// Open the CSV.
f, err := os.Open("myfile.csv")
if err != nil {
    log.Fatal(err)
}

// Read in the CSV records.
r := csv.NewReader(f)
records, err := r.ReadAll()
if err != nil {
    log.Fatal(err)
}

// Get the maximum value in the integer column.
var intMax int
for _, record := range records {

    // Parse the integer value.
    intVal, err := strconv.Atoi(record[0])
    if err != nil {
        log.Fatal(err)
    }

    // Replace the maximum value if appropriate.
    if intVal > intMax {
        intMax = intVal
    }
}

// Print the maximum value.
fmt.Println(intMax)
```

如果这个 CSV 文件的所有整型数值都存在，它将会产生同样正确的结果：

仅供非商业用途或交流学习使用

```
$ go build
$ ./myprogram
3
```

但是和之前的 Python 代码相比，Go 代码在 CSV 输入遇到非预期的情况下（例如在移除数值 3 的情况下），会给予用户提示：

```
$ go build
$ ./myprogram
2017/04/29 12:29:45 strconv.ParseInt: parsing "": invalid syntax
```

以上 Go 语言的程序维持了数据完整性，用户可以毫无顾虑地用一种适合自己的方法来处理数值丢失情况。

1.2 Go 语言收集和组织数据的最佳实践

正如前文所提及的，在数据收集、解析和组织的过程中，Go 语言可以维持高级别的数据完整性。在任何为机器学习工作流程准备数据的时候，都应该确保采用了 Go 语言的独特特性。

一般来说，使用 Go 语言的数据科学家 / 分析师在收集和组织数据的时候，应遵循以下最佳实践。这些最佳实践致力于维护应用中数据的完整性，并且使用户可以重现任何分析数据：

1. **检查并强制指定期望的类型**：这看起来可能是显而易见的，但是当使用动态类型语言的时候，却经常被忽视。所以虽然有些烦琐，但是明确地将数据解析为期望类型和明确地处理相关错误能解决使用过程中大量令人头疼的问题。
2. **标准化和简化数据输入 / 输出**：有许多第三方的程序包可用于处理特定类型的数据，或者同特定类型的数据源进行交互（本书会包含其中一部分）。但如果将与数据源的交互方式标准化，尤其是要以 `stdlib` 的使用为中心，就可以开发出可预测的模型，并且在团队中维持一致性。举例来说，应该选择 `database/sql` 同数据库进行交互，而不是选择各种各样第三方的 API 和 DSL。
3. **数据的版本控制**：不同的训练数据、选择的参数和输入的数据，会让机器学习模型产生极其不同的结果。因此，在没有对代码和数据进行版本控制的情况下，是不可能重现结果的。本章后面将会讨论数据版本管理的恰当技术。

一旦开始偏离这些常见的原则，就应该立刻停止手头的工作。因为可能会因为便利性而牺牲完整性，而这是一条危险的道路。在本书后续章节中涉及各式各样的数据格式 / 源时，会严格遵守这些原则。

仅供非商业用途或交流学习使用

1.3　CSV 文件

CSV 文件可能并不是大数据的首选格式，但是作为投身于机器学习的数据科学家或开发者而言，是一定会遇到这种格式的。用户可能需要一个邮编到经度 / 纬度的映射表，然后在因特网上找到对应的 CSV 文件，或者从销售团队获得 CSV 格式的销售数据。不管是哪种情况，都需要理解如何解析这种文件。

我们主要使用 `encoding/csv` 来解析 CSV 文件，这个软件包来自于 Go 语言标准库文件。即便如此，我们还是会讨论一些其他的软件包用以帮助读者快速操作和转换 CSV 数据，例如 `github.com/kniren/gota/dataframe` 和 `go-hep.org/x/hep/csvutil`。

1.3.1　从文件中读取 CSV 数据

以一个简单的 CSV 文件为例，该文件名称为 `iris.csv`（可以从此处下载：https://archive.ics.uci.edu/ml/datasets/iris）。这个 CSV 文件中的四列数值是浮点型，用以保存花朵的尺寸，还有一列是字符串型，用以保存花朵的品种：

```
$ head iris.csv
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
5.0,3.6,1.4,0.2,Iris-setosa
5.4,3.9,1.7,0.4,Iris-setosa
4.6,3.4,1.4,0.3,Iris-setosa
5.0,3.4,1.5,0.2,Iris-setosa
4.4,2.9,1.4,0.2,Iris-setosa
4.9,3.1,1.5,0.1,Iris-setosa
```

导入 `encoding/csv` 后，打开 CSV 文件然后创建该文件的读取函数：

```
// Open the iris dataset file.
f, err := os.Open("../data/iris.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a new CSV reader reading from the opened file.
reader := csv.NewReader(f)
```

随后可以从 CSV 文件中读取所有记录（对应行），这些记录被导出为 `[][]string`：

```
// Assume we don't know the number of fields per line. By setting
// FieldsPerRecord negative, each row may have a variable
// number of fields.
reader.FieldsPerRecord = -1
```

仅供非商业用途或交流学习使用

```
// Read in all of the CSV records.
rawCSVData, err := reader.ReadAll()
if err != nil {
    log.Fatal(err)
}
```

我们也可以在一个无限循环中每次读入一条记录，确保检查文件的结束符号（io.EOF），以保证在所有数据读取完毕后循环会退出。

```
// Create a new CSV reader reading from the opened file.
reader := csv.NewReader(f)
reader.FieldsPerRecord = -1

// rawCSVData will hold our successfully parsed rows.
var rawCSVData [][]string

// Read in the records one by one.
for {

    // Read in a row. Check if we are at the end of the file.
    record, err := reader.Read()
    if err == io.EOF {
        break
    }

    // Append the record to our dataset.
    rawCSVData = append(rawCSVData, record)
}
```

如果CSV文件不是以逗号进行分隔，或者文件中包含注释的行，那么可以利用csv.Reader.Comma和csv.Reader.Comment这两个域来处理这种特殊格式的CSV文件。如果CSV文件中包含单引号的域，那么可能需要添加帮助函数用以处理单引号和解析数值。

1.3.2 处理非预期的域

如果是干净的CSV数据，这种方法会处理得很好。但通常情况下很难遇到干净的数据，所以不得不解析脏数据。例如，可能会在CSV文件中碰到非预期的域，或者非预期数量的域。这便是reader.FieldsPerRecord存在的原因，这个读文件函数的值域可以让脏数据处理变得简单。如下所示：

```
4.3,3.0,1.1,0.1,Iris-setosa
5.8,4.0,1.2,0.2,Iris-setosa
5.7,4.4,1.5,0.4,Iris-setosa
5.4,3.9,1.3,0.4,blah,Iris-setosa
5.1,3.5,1.4,0.3,Iris-setosa
5.7,3.8,1.7,0.3,Iris-setosa
5.1,3.8,1.5,0.3,Iris-setosa
```

仅供非商业用途或交流学习使用

在这个版本的 iris.csv 文件里，其中一行有一个额外的域。假设每条记录应该包含 5 个域，那么就将 reader.FieldsPerRecord 的值设置为 5：

```
// We should have 5 fields per line. By setting
// FieldsPerRecord to 5, we can validate that each of the
// rows in our CSV has the correct number of fields.
reader.FieldsPerRecord = 5
```

然后从 CSV 文件中读取记录，检查非预期的域并且维护数据的完整性。

```
// rawCSVData will hold our successfully parsed rows.
var rawCSVData [][]string

// Read in the records looking for unexpected numbers of fields.
for {

    // Read in a row. Check if we are at the end of the file.
    record, err := reader.Read()
    if err == io.EOF {
        break
    }

    // If we had a parsing error, log the error and move on.
    if err != nil {
        log.Println(err)
        continue
    }

    // Append the record to our dataset, if it has the expected
    // number of fields.
    rawCSVData = append(rawCSVData, record)
}
```

在上面的代码中，可以通过记录错误日志来处理错误情况。在这里仅仅收集解析成功的记录，并将该记录存入 rawCSVData 中。读者会注意到这个错误可以用很多种不同的方式来处理。重要的是，读者应该严格要求自己检查数据是否为预期的属性，并且在应用中提高完整性。

1.3.3　处理非预期的类型

在上文中，CSV 的数据通过 Go 语言以 [][]string 类型读入。然而 Go 的数据类型是静态的，这让用户可以严格检查 CSV 文件的每个域。在后面的处理中，每个域都可以这么解析。考虑到有一些脏数据的域是随机的，在某一列中的脏数据无法和此列中其他数据的类型匹配：

```
4.6,3.1,1.5,0.2,Iris-setosa
5.0,string,1.4,0.2,Iris-setosa
5.4,3.9,1.7,0.4,Iris-setosa
5.3,3.7,1.5,0.2,Iris-setosa
5.0,3.3,1.4,0.2,Iris-setosa
```

仅供非商业用途或交流学习使用

```
7.0,3.2,4.7,1.4,Iris-versicolor
6.4,3.2,4.5,1.5,
6.9,3.1,4.9,1.5,Iris-versicolor
5.5,2.3,4.0,1.3,Iris-versicolor
4.9,3.1,1.5,0.1,Iris-setosa
5.0,3.2,1.2,string,Iris-setosa
5.5,3.5,1.3,0.2,Iris-setosa
4.9,3.1,1.5,0.1,Iris-setosa
4.4,3.0,1.3,0.2,Iris-setosa
```

为了检查CSV中域的类型，需要创建一个struct变量去保存成功解析出来的数值：

```
// CSVRecord contains a successfully parsed row of the CSV file.
type CSVRecord struct {
    SepalLength  float64
    SepalWidth   float64
    PetalLength  float64
    PetalWidth   float64
    Species      string
    ParseError   error
}
```

然后在遍历记录之前，初始化这些变量：

```
// Create a slice value that will hold all of the successfully parsed
// records from the CSV.
var csvData []CSVRecord
```

当遍历过所有的数据后，就可以解析记录中的相关类型，捕获任何错误，并在需要时记录日志：

```
// Read in the records looking for unexpected types.
for {

    // Read in a row. Check if we are at the end of the file.
    record, err := reader.Read()
    if err == io.EOF {
        break
    }

    // Create a CSVRecord value for the row.
    var csvRecord CSVRecord

    // Parse each of the values in the record based on an expected type.
    for idx, value := range record {

        // Parse the value in the record as a string for the string column.
        if idx == 4 {

            // Validate that the value is not an empty string. If the
            // value is an empty string break the parsing loop.
            if value == "" {
                log.Printf("Unexpected type in column %d\n", idx)
                csvRecord.ParseError = fmt.Errorf("Empty string value")
                break
            }
```

仅供非商业用途或交流学习使用

```
            // Add the string value to the CSVRecord.
            csvRecord.Species = value
            continue
        }

        // Otherwise, parse the value in the record as a float64.
        var floatValue float64

        // If the value can not be parsed as a float, log and break the
        // parsing loop.
        if floatValue, err = strconv.ParseFloat(value, 64); err != nil {
            log.Printf("Unexpected type in column %d\n", idx)
            csvRecord.ParseError = fmt.Errorf("Could not parse float")
            break
        }

        // Add the float value to the respective field in the CSVRecord.
        switch idx {
        case 0:
            csvRecord.SepalLength = floatValue
        case 1:
            csvRecord.SepalWidth = floatValue
        case 2:
            csvRecord.PetalLength = floatValue
        case 3:
            csvRecord.PetalWidth = floatValue
        }
    }

    // Append successfully parsed records to the slice defined above.
    if csvRecord.ParseError == nil {
        csvData = append(csvData, csvRecord)
    }
}
```

1.3.4　用数据帧操作 CSV 数据

如大家所见，手动解析不同的域，然后一行行地进行操作是非常烦琐和无味的。但是不能以此为由而增加系统的复杂性和导入一大批非标准的功能。大多数情况下应当默认使用 encoding/csv。

然而，数据帧的操作则被证明是一个成功的方法，在某些方面（比如在数据科学社区）甚至可以作为一种处理表格数据的标准方式。因此，在一些场景中，是值得使用第三方的函数去操作类似 CSV 的表格数据的。例如数据帧和相应的功能就非常适合于表格数据集合的过滤、求子集和数据的部分选择。本节将介绍 github.com/kniren/gota/dataframe，这是一个用于 Go 语言的优秀的 dataframe 程序包：

```
import "github.com/kniren/gota/dataframe"
```

为了从 CVS 文件中创建数据帧，这里使用 os.Open() 打开一个文件，然后将

仅供非商业用途或交流学习使用

返回的指针传递给 dataframe.ReadCSV() 函数：

```
// Open the CSV file.
irisFile, err := os.Open("iris.csv")
if err != nil {
    log.Fatal(err)
}
defer irisFile.Close()

// Create a dataframe from the CSV file.
// The types of the columns will be inferred.
irisDF := dataframe.ReadCSV(irisFile)

// As a sanity check, display the records to stdout.
// Gota will format the dataframe for pretty printing.
fmt.Println(irisDF)
```

如果编译和运行这个 Go 程序，会看到一个很棒的、打印良好的数据版本，这些数据的类型在解析中被自动推断出来：

```
$ go build
$ ./myprogram
[150x5] DataFrame

 sepal_length sepal_width petal_length petal_width species
 0: 5.100000 3.500000 1.400000 0.200000 Iris-setosa
 1: 4.900000 3.000000 1.400000 0.200000 Iris-setosa
 2: 4.700000 3.200000 1.300000 0.200000 Iris-setosa
 3: 4.600000 3.100000 1.500000 0.200000 Iris-setosa
 4: 5.000000 3.600000 1.400000 0.200000 Iris-setosa
 5: 5.400000 3.900000 1.700000 0.400000 Iris-setosa
 6: 4.600000 3.400000 1.400000 0.300000 Iris-setosa
 7: 5.000000 3.400000 1.500000 0.200000 Iris-setosa
 8: 4.400000 2.900000 1.400000 0.200000 Iris-setosa
 9: 4.900000 3.100000 1.500000 0.100000 Iris-setosa
 ... ... ... ... ...
 <float> <float> <float> <float> <string>
```

一旦所有的数据都被解析到一个 dataframe 中，便可以轻松地对数据进行过滤、求子集以及选择了：

```
// Create a filter for the dataframe.
filter := dataframe.F{
    Colname: "species",
    Comparator: "==",
    Comparando: "Iris-versicolor",
}

// Filter the dataframe to see only the rows where
// the iris species is "Iris-versicolor".
versicolorDF := irisDF.Filter(filter)
if versicolorDF.Err != nil {
    log.Fatal(versicolorDF.Err)
}

// Filter the dataframe again, but only select out the
```

仅供非商业用途或交流学习使用

```
// sepal_width and species columns.
versicolorDF = irisDF.Filter(filter).Select([]string{"sepal_width",
"species"})

// Filter and select the dataframe again, but only display
// the first three results.
versicolorDF = irisDF.Filter(filter).Select([]string{"sepal_width",
"species"}).Subset([]int{0, 1, 2})
```

以上仅仅只是触碰到 github.com/kniren/gota/dataframe 的皮毛。这个程序包可以合并数据集合，输出为其他格式，甚至处理 JSON 类型的数据。如果想要了解更多信息，读者可以参考自动生成的 Go 文档，https://godoc.org/github.com/kniren/gota/dataframe，这也是一个最佳实践，一般来说，本书中讨论的任何程序包都可以这么做。

1.4　JSON

在当今社会，绝大多数的数据都是通过网络存取的，并且很多工程机构也实施了一些微服务，因此我们经常能遇到 JSON 格式的数据。我们可能是通过 API 抓取一些随机数据时需要处理这种数据格式，也有可能将它作为主要的数据格式用以驱动数据分析和机器学习的流程。

如果将易用性作为数据交换的主要目标，那么 JSON 是非常适合的。因为 JSON 格式具有可读性，如果某些地方出问题了可以很方便地进行调试。当用 Go 处理数据时，需要记住我们的目标是在数据处理过程中维护完整性，而这个过程中的一部分是，尽量保持数据的可读性和可翻译性。在实现这些目标的时候，JSON 被发现是非常有用的（这是在许多情况下，JSON 也用于记录日志的原因）。

Go 在它的标准库中以 encoding/json 的方式提供 JSON 功能，本书将全部采用这种标准库函数方式。

1.4.1　JSON 的解析

为了理解如何在 Go 中解析（也就是解组）JSON 数据，需要使用一些来自 Citi Bike API 的数据（https://www.citibikenyc.com/system-data），这是一个位于纽约的单车共享服务。Citi Bike 以 JSON 格式提供了它们的自行车共享站点网络中经常更新的操作信息。这些信息位于 https://gbfs.citibikenyc.com/gbfs/en/station_status.json：

```
{
  "last_updated": 1495252868,
  "ttl": 10,
  "data": {
    "stations": [
```

仅供非商业用途或交流学习使用

```
        {
          "station_id": "72",
          "num_bikes_available": 10,
          "num_bikes_disabled": 3,
          "num_docks_available": 26,
          "num_docks_disabled": 0,
          "is_installed": 1,
          "is_renting": 1,
          "is_returning": 1,
          "last_reported": 1495249679,
          "eightd_has_available_keys": false
        },

        {
          "station_id": "79",
          "num_bikes_available": 0,
          "num_bikes_disabled": 0,
          "num_docks_available": 33,
          "num_docks_disabled": 0,
          "is_installed": 1,
          "is_renting": 1,
          "is_returning": 1,
          "last_reported": 1495248017,
          "eightd_has_available_keys": false
        },

        etc...

        {
          "station_id": "3464",
          "num_bikes_available": 1,
          "num_bikes_disabled": 3,
          "num_docks_available": 53,
          "num_docks_disabled": 0,
          "is_installed": 1,
          "is_renting": 1,
          "is_returning": 1,
          "last_reported": 1495250340,
          "eightd_has_available_keys": false
        }
      ]
    }
}
```

要解析 Go 中导入的数据和类似这种类型的数据，首先需要导入 encoding/json（以及标准库中的其他一些东西，例如 net/http，因为需要从前面提到的网站下载数据）。还将定义模拟上述代码中 JSON 结构的 struct：

```
import (
    "encoding/json"
    "fmt"
    "io/ioutil"
    "log"
    "net/http"
)

// citiBikeURL provides the station statuses of CitiBike bike sharing
```

仅供非商业用途或交流学习使用

```
stations.
const citiBikeURL =
"https://gbfs.citibikenyc.com/gbfs/en/station_status.json"

// stationData is used to unmarshal the JSON document returned form
citiBikeURL.
type stationData struct {
    LastUpdated int `json:"last_updated"`
    TTL int `json:"ttl"`
    Data struct {
        Stations []station `json:"stations"`
    } `json:"data"`
}

// station is used to unmarshal each of the station documents in
stationData.
type station struct {
    ID string `json:"station_id"`
    NumBikesAvailable int `json:"num_bikes_available"`
    NumBikesDisabled int `json:"num_bike_disabled"`
    NumDocksAvailable int `json:"num_docks_available"`
    NumDocksDisabled int `json:"num_docks_disabled"`
    IsInstalled int `json:"is_installed"`
    IsRenting int `json:"is_renting"`
    IsReturning int `json:"is_returning"`
    LastReported int `json:"last_reported"`
    HasAvailableKeys bool `json:"eightd_has_available_keys"`
}
```

有一些事情需要注意：这里已经遵守 Go 的语法规范，避免使用包含下划线的 `struct` 域名，但是这里已经使用了 `json` 的结构标签，在 JSON 数据中将 `struct` 域标记为期望值。

请注意，为了正确解析 JSON 数据，结构域应该是能被导出的。也就是说，域的名称要以大写字母开头。`encoding/json` 无法通过反射的方式查看域，除非这些域被导出。

现在可以从 URL 中获取到 JSON 数据并解组成一个新的 `stationData` 值。这会生成一个 `struct` 变量，其中各自的域值由被标记的 JSON 数据域来填充。可以通过打印出与某个车站相关联的数据进行核对：

```
// Get the JSON response from the URL.
response, err := http.Get(citiBikeURL)
if err != nil {
    log.Fatal(err)
}
defer response.Body.Close()

// Read the body of the response into []byte.
body, err := ioutil.ReadAll(response.Body)
if err != nil {
    log.Fatal(err)
}
```

```
// Declare a variable of type stationData.
var sd stationData

// Unmarshal the JSON data into the variable.
if err := json.Unmarshal(body, &sd); err != nil {
    log.Fatal(err)
}

// Print the first station.
fmt.Printf("%+v\n\n", sd.Data.Stations[0])
```

当运行这个程序的时候，可以看到返回的 struct 中包含从 URL 中解析的数据：

```
$ go build
$ ./myprogram
{ID:72 NumBikesAvailable:11 NumBikesDisabled:0 NumDocksAvailable:25
NumDocksDisabled:0 IsInstalled:1 IsRenting:1 IsReturning:1
LastReported:1495252934 HasAvailableKeys:false}
```

1.4.2 JSON 的输出

现在已经从 stationData 结构体的取值中得到了 Citi Bike 的站点数据，如果要把这些数据存储在文件中，可以通过修改 json.marshal 文件来实现：

```
// Marshal the data.
outputData, err := json.Marshal(sd)
if err != nil {
    log.Fatal(err)
}

// Save the marshalled data to a file.
if err := ioutil.WriteFile("citibike.json", outputData, 0644); err != nil {
    log.Fatal(err)
}
```

1.5 SQL-like 数据库

虽然对于有趣的 NoSQL 数据库和键值（key-value）存储有一些炒作，但 SQL-like 数据库仍然是普遍存在的。在某些时候，每个数据科学家都将会在某种情况下，处理来自 SQL-like 数据库中的数据，例如 Postgres、MySQL 或者 SQLite。

比如可能需要查询 Postgres 数据库的一个或一组表用于生成模型训练的特征集。在使用该模型进行预测或识别异常之后，可能会将结果发送到另一个数据库的表中，用以驱动仪表盘或者其他报告工具。

当然，Go 语言可以非常友好地与所有流行的数据存储进行交互，例如 SQL、NoSQL、键值等。但是这里，重点是放在 SQL-Like 的交互上。本书全都会采用 databse/sql 作为数据库的交互方式。

1.5.1　连接到一个 SQL 数据库

在连接到一个 SQL-like 数据库之前，首先要识别将要与之交互的数据库类型并导入相应的驱动程序。在以下示例中，会连接到一个 Postgres 数据库，并使用 `github.com/lib/pq` 作为 `database/sql` 的数据库驱动。这个驱动可以通过空导入进行加载（带有相应的注释）：

```
import (
    "database/sql"
    "fmt"
    "log"
    "os"

    // pq is the library that allows us to connect
    // to postgres with databases/sql.
    _ "github.com/lib/pq"
)
```

现在假设已经把 Postgres 连接字符串导出到一个环境变量 `PGURL` 中。可以很容易地通过如下代码为数据库连接创建一个 `sql.DB` 值：

```
// Get the postgres connection URL. I have it stored in
// an environmental variable.
pgURL := os.Getenv("PGURL")
if pgURL == "" {
    log.Fatal("PGURL empty")
}

// Open a database value. Specify the postgres driver
// for databases/sql.
db, err := sql.Open("postgres", pgURL)
if err != nil {
    log.Fatal(err)
}
defer db.Close()
```

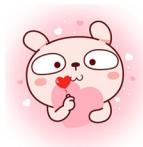
请注意需要在这个程序中对 `close` 方法使用延时函数（`defer`）。另外还需注意创建此值并不意味着成功连接到了数据库。这仅仅是 `database/sql` 执行某些操作（比如查询）触发数据库连接而使用的值。

为了确保能够成功连接到数据库，可以使用 `Ping` 方法核实：

```
if err := db.Ping(); err != nil {
    log.Fatal(err)
}
```

1.5.2　查询数据库

现在已经知道了如何连接数据库，接下来看一下如何从数据库中获取数据。在本书中不会介绍 SQL 查询和 SQL 语法相关的知识。如果读者并不熟悉 SQL 语言，



强烈建议学习一下 query、insert 等 SQL 语句的相关知识。在本书的学习中，这两种最基本的 SQL 操作是必会的，并会在后续章节中用来访问 SQL 数据库：

- 一个 Query 操作，用于在数据库中选择、分组、汇总数据并返回数据行。
- 一个 Exec 操作，用于数据库的更新、插入，或者更改其状态，这些操作无须返回数据库中存储的部分数据。

正如你所期望的，将使用 Query 操作从数据库中获取数据。为此，需要使用一个 SQL 语句字符串去查询数据库。比如，假设已有一个数据库用于存储一批鸢尾花（iris flower）的测量数据（包括花瓣长度、花瓣宽度等），那么就可以如下所示查询一些跟特定鸢尾花品种相关的数据：

```
// Query the database.
rows, err := db.Query(`
    SELECT
        sepal_length as sLength,
        sepal_width as sWidth,
        petal_length as pLength,
        petal_width as pWidth
    FROM iris
    WHERE species = $1`, "Iris-setosa")
if err != nil {
    log.Fatal(err)
}
defer rows.Close()
```

注意，这里会返回一个指向 sql.Rows 值的指针，需要在 defer 中定义这行值的关闭操作。然后循环遍历返回的行，将数据解析为期望类型。在返回行中使用 Scan 方法，来解析出 SQL 查询所返回的列，并将结果打印到标准输出中：

```
// Iterate over the rows, sending the results to
// standard out.
for rows.Next() {

    var (
        sLength float64
        sWidth float64
        pLength float64
        pWidth float64
    )

    if err := rows.Scan(&sLength, &sWidth, &pLength, &pWidth); err != nil {
        log.Fatal(err)
    }

    fmt.Printf("%.2f, %.2f, %.2f, %.2f\n", sLength, sWidth, pLength,
pWidth)
}
```

最后，需要检查在处理数据行的时候，任何可能发生的错误。这么做是为了保证数据处理的可靠性，因为不能假定遍历数据时会一帆风顺：

```
// Check for errors after we are done iterating over rows.
if err := rows.Err(); err != nil {
    log.Fatal(err)
}
```

1.5.3　修改数据库

如前所述，使用 Exec 操作数据库是另外一种风格。使用这种类型的语句时，更需要关心的是更新、添加或者修改数据库中一个或多个表的状态。这里使用相同类型的数据库连接，但是调用的是 db.Exec 而不是 db.Query。

例如，假设要更新鸢尾花数据库表中的一些值：

```
// Update some values.
res, err := db.Exec("UPDATE iris SET species = 'setosa' WHERE species =
'Iris-setosa'")
if err != nil {
    log.Fatal(err)
}
```

但是怎么知道已经成功完成了修改的操作呢？利用这里返回的 res 函数，可以得知数据表中有多少行受到更新操作的影响：

```
// See how many rows where updated.
rowCount, err := res.RowsAffected()
if err != nil {
    log.Fatal(err)
}

// Output the number of rows to standard out.
log.Printf("affected = %d\n", rowCount)
```

1.6　缓存

有时，用于产生预测的机器学习算法，其用作训练的输入可能来自外部数据源（例如各种 API），也就是说，用于建模或者分析的应用程序，并没有使用本地的数据。进一步来讲，可能有各种各样被频繁访问的数据集，这些数据可能很快会被再次访问，或者在应用程序运行中需要保持可用。

至少在某些情况下，将数据缓存到内存中或者将数据嵌入本地应用程序运行的环境中是非常有意义的。例如，如果经常需要访问政府 API（通常来说是高延时的）用以获取人口普查的数据，应当考虑在本地或者内存中缓存一份普查数据，用来避免不断地访问该 API。

1.6.1　在内存中缓存数据

在这里将使用 github.com/patrickmn/go-cache，以在内存中缓存一系列

数值。使用这个软件包，可以创建一个在内存中的主键与对应值的缓存。甚至可以做一些特定操作，比如针对缓存中指定的键值对设置生存时间。

要创建一个新的内存缓存并在缓存中设置一个键值对，需要执行如下操作：

```
// Create a cache with a default expiration time of 5 minutes, and which
// purges expired items every 30 seconds
c := cache.New(5*time.Minute, 30*time.Second)

// Put a key and value into the cache.
c.Set("mykey", "myvalue", cache.DefaultExpiration)
```

然后只需使用 Get 方法，便可将 mykey 的值从缓存中检索出来：

```
v, found := c.Get("mykey")
if found {
    fmt.Printf("key: mykey, value: %s\n", v)
}
```

1.6.2 在本地磁盘中缓存数据

刚刚看到的缓存是位于内存中的。也就是说，只有应用程序在运行时，缓存的数据才存在并且能被访问，一旦应用程序退出，数据就会消失。在某些情况下，当应用程序重新启动或者退出时，仍然希望缓存的数据能够访问。我们可能还要备份缓存，这样就能避免在没有相关数据缓存的情况下从头开始启动应用程序。

在这些情况下，可能需要考虑使用本地的、嵌入式的缓存，例如 github.com/boltdb/bolt。下面要谈到的 BoltDB 就是深受这类应用所欢迎的一个项目，这个项目基本是由本地键值存储构成的。请执行以下操作来初始化其中一个键值数据仓库：

```
// Open an embedded.db data file in your current directory.
// It will be created if it doesn't exist.
db, err := bolt.Open("embedded.db", 0600, nil)
if err != nil {
    log.Fatal(err)
}
defer db.Close()

// Create a "bucket" in the boltdb file for our data.
if err := db.Update(func(tx *bolt.Tx) error {
    _, err := tx.CreateBucket([]byte("MyBucket"))
    if err != nil {
        return fmt.Errorf("create bucket: %s", err)
    }
    return nil
}); err != nil {
    log.Fatal(err)
}
```

当然，可以在 BoltDB 中使用多个不同的数据桶，并且使用文件名而不是 embeded.db。

接下来，假设在内存中已经存在一个字符串的映射，这个映射需要被缓存到 BoltDB 中。为了实现这个功能，需要遍历映射中的键和值，并更新到 BoltDB 中：

```
// Put the map keys and values into the BoltDB file.
if err := db.Update(func(tx *bolt.Tx) error {
    b := tx.Bucket([]byte("MyBucket"))
    err := b.Put([]byte("mykey"), []byte("myvalue"))
    return err
}); err != nil {
    log.Fatal(err)
}
```

然后，要从 BoltDB 中获取值，可以通过如下方法查看数据：

```
// Output the keys and values in the embedded
// BoltDB file to standard out.
if err := db.View(func(tx *bolt.Tx) error {
    b := tx.Bucket([]byte("MyBucket"))
    c := b.Cursor()
    for k, v := c.First(); k != nil; k, v = c.Next() {
        fmt.Printf("key: %s, value: %s\n", k, v)
    }
    return nil
}); err != nil {
    log.Fatal(err)
}
```

1.7　数据版本控制

如上所述，机器学习模型会随着使用的训练数据、选择的参数和输入数据的不同产生非常不同的结果。为了具有协作性、创造性和合规性，重现结果是至关重要的。

- **协作性**：尽管在社交媒体上会看到夸张的人物宣传，但是没有数据科学和机器学习独角兽（即同时具备数据科学和机器学习各领域的知识和能力的人）。这项工作需要同僚们的共同评审和持续改进，如果不能再现模型的结果和分析，这样的结果是不可取的。
- **创造性**：在协同工作中我们互相并不熟知，而且也很难记住哪怕是昨天做的每一件事。不能认为自己会永远记住之前所有的推理和逻辑，特别是当处理机器学习工作流程的时候。使用的数据、创造的结果以及创造的方法都需要被精确地追踪。这是模型和技术能够不断改进的唯一方式。
- **合规性**：最后，机器学习中的数据版本控制和可重现性很快就会成为必选项。法律正在世界各地传播（例如欧盟的**一般数据保护条例**（GDPR，General Data Protection Regulation）），使用户有权得到算法做出的决定的解释。如果

仅供非商业用途或交流学习使用

没有可靠的方式跟踪正在处理的数据以及正在生产的结果，机器学习将根本无法遵守这些条例。

版本控制有很多个开源项目。其中一些专注于数据的安全性和点对点分布式存储，另一些则专注于数据科学工作流程。本书将重点关注和利用 Pachyderm（http://pachyderm.io/），这是用于数据版本控制和数据流水线的开源框架。这么做的原因稍后会在讲述生产部署和 ML 流水线管理中提及。现在，将总结 Pachyderm 的一些功能，它是基于 Go（和其他语言）的 ML 项目的数据版本控制程序，这些功能使选择 Pachyderm 时更加有吸引力：

- 它有一个方便的 Go 客户端，`github.com/pachyderm/pachyderm/src/client`
- 能够对任何数据类型和格式进行版本化
- 支持版本化数据的灵活的对象存储
- 与数据流水线系统集成，用于驱动版本化的 ML 工作流程

1.7.1 Pachyderm 术语

Pachyderm 中的数据版本控制和 Git 中的代码版本控制类似。很多基础概念是相似的：

- **数据仓库**：这些是版本化的数据集合，类似于 Git 数据仓库中版本化的代码集合
- **提交**：数据通过提交进入到数据仓库中以在 Pachyderm 中进行版本控制
- **分支**：这些轻量级的点指向一个特定提交，或者一批特定提交（例如 master 指向最新的 HEAD 提交）
- **文件**：在 Pachyderm 中，数据是以文件级别进行版本控制的。并且 Pachyderm 自动采用诸如去重的策略来保持版本化数据空间的高效性

即使 Pachyderm 的版本控制数据与 Git 的版本控制代码类似，也会有一些主要的区别。例如，合并数据并不完全一致。如果在数 P 字节（petabyte）数据上存在合并冲突，那么没有人可以解决这些问题。此外，Git 协议在处理大型数据集合上，空间利用并不高效。Pachyderm 使用自己的内部逻辑来执行版本控制并使用版本化数据，其处理逻辑在缓存方面具有高空间利用率和高处理效率。

1.7.2 部署 / 安装 Pachyderm

本书中的其他很多地方将使用 Pachyderm 进行数据版本控制和创建分布式 ML

仅供非商业用途或交流学习使用

工作流程。Pachyderm 本身是在 Kubernetes（https://kubernetes.io/）之上运行的应用程序，并且提供对象仓库作为选择。可以在本地非常容易地安装和运行 Pachyderm 以用于本书所涉及的开发和实现。安装需要 5～10 分钟，不需要太多的工作量。本地安装的说明可以在 Pachyderm 文档中找到：http://docs.pachyderm.io 。

当准备好在生产环境或部署模型中运行工作流程时，也可以很容易地部署一个服务于生产环境的 Pachyderm 集群，其行为与本地安装的 Pachyderm 完全相同。Pachyderm 可以部署到任何云端，甚至可以部署在企业预置场景中。

如上所述，Pachyderm 是一个开源项目，拥有一个活跃的用户群体。如果你有任何问题需要帮助，可以通过访问 http://slack.pachyderm.io 来加入公共的 Pachyderm Slack 频道。活跃的 Pachyderm 用户和 Pachyderm 团队将能够快速响应你的问题。

1.7.3　创建用于数据版本控制的数据仓库

如果按照 Pachyderm 文档中指定的 Pachyderm 本地安装方式，会找到如下内容：

- 运行于本机 Minikube VM 中的 Kubernetes
- 安装并连接到 Pachyderm 集群的 `pachctl` 命令行工具

当然，如果你使用的是云上的产品集群，还需要如下步骤。`pachctl` 命令行工具会连接到远程集群。

如下所示的 `pachctl` 命令行界面（CLI，Command-Line Interface）工具（这是一个用 Go 语言写的程序）用来展示数据版本控制功能。然而就像之前所讲，Pachyderm 有一个成熟的 Go 客户端。可以用 Go 程序直接创建数据仓库、提交数据。这部分功能会在第 9 章中进行说明。

运行如下代码，可创建名为 `myrepo` 的数据仓库：

```
$ pachctl create-repo myrepo
```

然后，可以使用 `list-repo` 来确认数据仓库是否已经创建：

```
$ pachctl list-repo
NAME CREATED SIZE
myrepo 2 seconds ago 0 B
```

这个 `myrepo` 数据仓库是已经定义的数据集合，并且已经准备好存放版本化的数据。现在，因为还没有添加任何数据，所以该数据仓库仍然是空的。

1.7.4　把数据存储到数据仓库中

这里有一个简单的文本文件：

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

```
$ cat blah.txt
This is an example file.
```

如果这个文件是机器学习工作流所使用数据的一部分，则需要对其版本化。为了在 myrepo 数据仓库中将文件版本化，只需要将其提交到数据仓库中：

```
$ pachctl put-file myrepo master -c -f blah.txt
```

-c 标志表示 Pachyderm 一步做完三个动作：发起一个新的提交，上传正在引用的文件，关闭这个提交。-f 标志表示所传参数是一个文件。

请注意，现在正在提交单个文件到一个数据仓库的 master 分支。然而 Pachyderm API 是非常灵活的。可以在单次提交操作或多次提交操作中完成提交、删除甚至修改多个版本化文件。更进一步，这些文件可以通过 URL、对象存储链接、数据库转储等方式做版本化操作。

可以通过确认文件是否版本化于数据仓库中来进行完整性检查：

```
$ pachctl list-repo
NAME CREATED SIZE
myrepo 10 minutes ago 25 B
$ pachctl list-file myrepo master
NAME TYPE SIZE
blah.txt file 25 B
```

1.7.5 从版本化的数据仓库中获取数据

现在已经在 Pachyderm 中对数据完成了版本化，我们有可能想知道如何与数据进行交互。交互的主要途径是通过 Pachyderm 数据流水线（将在本书后面进行介绍）。使用流水线与版本化数据做交互的机制是简单的文件 I/O。

然而如果要手动地从 Pachyderm 中提取某些版本化数据集，以交互的方式对之进行分析，则可以通过 pathctl CLI 来获取数据：

```
$ pachctl get-file myrepo master blah.txt
This is an example file.
```

1.8 参考书目

CVS 数据：

- encoding/csv 文档：https://golang.org/pkg/encoding/csv/
- github.com/kniren/gota/dataframe 文档：https://godoc.org/github.com/kniren/gota/dataframe

JSON 数据：

- encoding/json 文档：https://golang.org/pkg/encoding/json/

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

- Bill Kennedy 的关于 JSON decoding 的博客：https://www.goinggo.net/2014/01/decode-json-documents-in-go.html
- Ben Johnson 的关于 Go Walkthrough 的博客：`encoding/json` package：https://medium.com/go-walkthrough/go-walkthrough-encoding-json-package-9681d1d37a8f

缓存：

- `github.com/patrickmn/go-cache` 文档：https://godoc.org/github.com/patrickmn/go-cache
- `github.com/boltdb/bolt` 文档：https://godoc.org/github.com/boltdb/bolt
- BoltDB 相关信息和使用该数据库的原因：https://npf.io/2014/07/intro-to-boltdb-painless-performant-persistence/

Pachyderm：

- 通用文档：http://docs.pachyderm.io
- Go 客户端文档：https://godoc.org/github.com/pachyderm/pachyderm/src/client
- Slack 团队公开用户注册：http://docs.pachyderm.io

1.9 小结

在这一章中涉及了如何收集、组织和解析数据。这是开发机器学习模型的第一步，也是最重要的步骤之一。但是如果没有从数据中获得一些直觉，并将其放入标准的处理模式中，那么数据的奥秘就不会展现。接下来，我们会攻克一些技术难题，包括进一步结构化数据（矩阵）和进一步理解数据（统计学和概率论）。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

第 2 章
矩阵、概率论和统计学

本书会着重介绍机器学习中最实际、应用最广的技术和方法。但不可避免，也需要介绍一些便于理解和易于应用的基本理论。尤其是一些有关概率论和统计学的理论，这些知识会帮助我们在遇到相关问题时匹配特定算法、理解数据和结果、对数据应用必要的转换。矩阵和一些线性代数的知识将有助于以恰当方式展示数据，实际优化、最小化以及基于矩阵的转换。

如果读者在数学或统计学方面有些生疏，也不用太过担心。本书会介绍一些基本理论，并且演示如何用后文提到的相关的统计方法和矩阵技术进行编码工作。也就是说，本书并不是关于统计学、概率论和线性代数的书籍。为了真正精通机器学习，读者还是应当花些时间深入学习这些主题。

2.1 矩阵和向量

如果在机器学习领域投入了大量时间学习和实践，会发现其中用到很多的矩阵和向量。事实上，许多机器学习算法归根到底就是一系列矩阵的迭代操作。矩阵和向量是什么？如何在 Go 程序中表示它们呢？

大部分时候，本书会用 `github.com/gonum` 提供的软件包去构建和处理矩阵以及向量。这批丰富的用于处理数值计算的 Go 语言软件包正在逐步改进并越来越完善。

2.1.1 向量

向量是指以行（从左向右）或者列（从上向下）方式排列的有序数字集合。向量中的每个数字称为一个元素。例如，它可能是一批代表公司销售情况的数字，或者

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

可能是一批代表温度的数字。

自然而然能想到利用 Go 进行数据切片，以展示这些有序的数据集合，如下所示。

```
// Initialize a "vector" via a slice.
var myvector []float64

// Add a couple of components to the vector.
myvector = append(myvector, 11.0)
myvector = append(myvector, 5.2)

// Output the results to stdout.
fmt.Println(myvector)
```

这些切片数据确实是有序的集合。然而它们并不真正表示行 / 列的概念，仍然需要在切片数据上进行各种各样的向量操作。所幸的是，在向量操作方面，gonum 提供了 gonum.org/v1/gonum/floats 用于操作 float64 类型的切片数据。在矩阵方面，gonum 在 gonum.org/v1/gonum/mat 中提供了 Vector 类型（以及相应的函数）。

```
// Create a new vector value.
myvector := mat.NewVector(2, []float64{11.0, 5.2})
```

2.1.2　向量操作

正如这里提到的，处理向量需要使用某些向量 / 矩阵的特殊操作和规则。例如，多个矩阵如何相乘？如何判断两个矩阵相似？ gonum.org/v1/gonum/floats 和 gonum.org/v1/gonum/mat 都提供了内置的方法和函数用于向量 / 切片操作。例如点积、排序、距离。函数库的内容非常丰富，这里不会覆盖全部的函数。但是可以大概知道应当如何使用向量运算。首先，可以像下面这样使用 gonum.org/v1/gonum/floats：

```
// Initialize a couple of "vectors" represented as slices.
vectorA := []float64{11.0, 5.2, -1.3}
vectorB := []float64{-7.2, 4.2, 5.1}

// Compute the dot product of A and B
// (https://en.wikipedia.org/wiki/Dot_product).
dotProduct := floats.Dot(vectorA, vectorB)
fmt.Printf("The dot product of A and B is: %0.2f\n", dotProduct)

// Scale each element of A by 1.5.
floats.Scale(1.5, vectorA)
fmt.Printf("Scaling A by 1.5 gives: %v\n", vectorA)

// Compute the norm/length of B.
normB := floats.Norm(vectorB, 2)
fmt.Printf("The norm/length of B is: %0.2f\n", normB)
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

使用 gonum.org/v1/gonum/mat 也可做类似操作：

```
// Initialize a couple of "vectors" represented as slices.
vectorA := mat.NewVector(3, []float64{11.0, 5.2, -1.3})
vectorB := mat.NewVector(3, []float64{-7.2, 4.2, 5.1})

// Compute the dot product of A and B
// (https://en.wikipedia.org/wiki/Dot_product).
dotProduct := mat.Dot(vectorA, vectorB)
fmt.Printf("The dot product of A and B is: %0.2f\n", dotProduct)

// Scale each element of A by 1.5.
vectorA.ScaleVec(1.5, vectorA)
fmt.Printf("Scaling A by 1.5 gives: %v\n", vectorA)

// Compute the norm/length of B.
normB := blas64.Nrm2(3, vectorB.RawVector())
fmt.Printf("The norm/length of B is: %0.2f\n", normB)
```

这两个方法的语法很类似。如果仅仅用到向量（不涉及矩阵），并且只需要针对浮点型数字切片进行一些轻量级和快速的操作，那么 gonum.org/v1/gonum/floats 会是一个很好的选择。然而，如果需要同时使用矩阵和向量，并且需要访问许多向量 / 矩阵函数，那么使用 gonum.org/v1/gonum/mat 会更明智（同时偶尔会参考 gonum.org/v1/gonum/blas/blas64）。

2.1.3 矩阵

矩阵和线性代数对许多人而言可能有点复杂，但是简单来说，矩阵就是一些像矩形般组织的数据，而线性代数就是定义它们操作的规则。例如，如下所示是一个 4×4 阵 A：

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{23} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

A 的组成元素（a_{11}、a_{12}，等等）是一些放入矩阵中的独立数字，这些元素的下标指明了元素在矩阵中的位置。第一个索引是行索引，第二个索引是列索引。通常来讲，M 行 N 列的矩阵 A 可以有任意的形状 / 大小：

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & \vdots & \vdots & \vdots \\ a_{M1} & a_{M2} & \cdots & a_{MN} \end{pmatrix}$$

为了用 gonum.org/v1/gonum/mat 生成一个类似矩阵，需要创建一份 float64 类型的切片用以代表所有的矩阵元素。在本例中，生成如下的矩阵：

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

$$\begin{pmatrix} 1.2 & -5.7 \\ -2.4 & 7.3 \end{pmatrix}$$

我们需要生成一份如下的 float64 类型切片：

```
// Create a flat representation of our matrix.
components := []float64{1.2, -5.7, -2.4, 7.3}
```

然后，把切片和维度信息提供给 gonum.org/v1/gonum/mat，用以生成新的 mat.Dense 矩阵：

```
// Form our matrix (the first argument is the number of
// rows and the second argument is the number of columns).
a := mat.NewDense(2, 2, data)

// As a sanity check, output the matrix to standard out.
fa := mat.Formatted(a, mat.Prefix(" "))
fmt.Printf("mat = %v\n\n", fa)
```

请注意这里利用了 gonum.org/v1/gonum/mat 良好的格式化逻辑来打印矩阵以进行完整性检查。运行后可以看到如下信息：

```
$ go build
$ ./myprogram
A = [ 1.2 -5.7]
    [-2.4  7.3]
```

然后可以通过一些内置的函数访问和修改 A 的特定数据：

```
// Get a single value from the matrix.
val := a.At(0, 1)
fmt.Printf("The value of a at (0,1) is: %.2f\n\n", val)

// Get the values in a specific column.
col := mat.Col(nil, 0, a)
fmt.Printf("The values in the 1st column are: %v\n\n", col)

// Get the values in a kspecific row.
row := mat.Row(nil, 1, a)
fmt.Printf("The values in the 2nd row are: %v\n\n", row)

// Modify a single element.
a.Set(0, 1, 11.2)

// Modify an entire row.
a.SetRow(0, []float64{14.3, -4.2})

// Modify an entire column.
a.SetCol(0, []float64{1.7, -0.3})
```

2.1.4　矩阵操作

与向量类似，矩阵有一套自己的算法规则和一整套特殊操作。一些跟矩阵有关

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

的算法能以相似的方法达到用户期望。但是进行矩阵相乘或者矩阵转置的时候，仍需格外小心。

为了方便起见，gonum.org/v1/gonum/mat 提供了一个用于算法和其他特殊操作的易用的 API。如下是一些操作的简单演示，例如加法、乘法、除法等：

```
// Create two matrices of the same size, a and b.
a := mat.NewDense(3, 3, []float64{1, 2, 3, 0, 4, 5, 0, 0, 6})
b := mat.NewDense(3, 3, []float64{8, 9, 10, 1, 4, 2, 9, 0, 2})

// Create a third matrix of a different size.
c := mat.NewDense(3, 2, []float64{3, 2, 1, 4, 0, 8})

// Add a and b.
d := mat.NewDense(0, 0, nil)
d.Add(a, b)
fd := mat.Formatted(d, mat.Prefix("            "))
fmt.Printf("d = a + b = %0.4v\n\n", fd)

// Multiply a and c.
f := mat.NewDense(0, 0, nil)
f.Mul(a, c)
ff := mat.Formatted(f, mat.Prefix("          "))
fmt.Printf("f = a c = %0.4v\n\n", ff)

// Raising a matrix to a power.
g := mat.NewDense(0, 0, nil)
g.Pow(a, 5)
fg := mat.Formatted(g, mat.Prefix("          "))
fmt.Printf("g = a^5 = %0.4v\n\n", fg)

// Apply a function to each of the elements of a.
h := mat.NewDense(0, 0, nil)
sqrt := func(_, _ int, v float64) float64 { return math.Sqrt(v) }
h.Apply(sqrt, a)
fh := mat.Formatted(h, mat.Prefix("              "))
fmt.Printf("h = sqrt(a) = %0.4v\n\n", fh)
```

请特别留意上面的 Apply() 方法，这个功能极其好用，因为它可以让用户对矩阵元素应用任何函数。可以对所有元素应用相同的函数，或者针对矩阵元素的下标选择性应用函数。例如，可以利用这个方法实现针对每个元素的相乘、用户定义函数的应用、第三方软件包的函数的应用。

其次，gonum.org/v1/gonum/mat 涵盖了所有功能，例如行列式、特征值/向量求解、矩阵转置。再次提示，本书不会针对所有的功能展开讨论，但是会演示一些简单的操作，如下所示：

```
// Create a new matrix a.
a := mat.NewDense(3, 3, []float64{1, 2, 3, 0, 4, 5, 0, 0, 6})

// Compute and output the transpose of the matrix.
ft := mat.Formatted(a.T(), mat.Prefix(" "))
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
fmt.Printf("a^T = %v\n\n", ft)

// Compute and output the determinant of a.
deta := mat.Det(a)
fmt.Printf("det(a) = %.2f\n\n", deta)

// Compute and output the inverse of a.
aInverse := mat.NewDense(0, 0, nil)
if err := aInverse.Inverse(a); err != nil {
    log.Fatal(err)
}
fi := mat.Formatted(aInverse, mat.Prefix(" "))
fmt.Printf("a^-1 = %v\n\n", fi)
```

请留意在本示例中，当需要确保维持完整性和可读性的时候，利用了 Go 语言显式的错误处理函数。矩阵并不是总能转置。在处理矩阵和大数据集的时候，经常会遇到各种各样类似上文的异常，我们需要确保应用程序如预期那样工作。

2.2　统计学

在最后，机器学习应用的成功与否取决于数据的质量、对数据的理解以及对结果的评估和校验。所有这三方面都要求对统计学有一定的理解。

利用统计学领域的知识可以获得对数据的理解，并且量化数据和结果。同时这也提供一些机制用以测量应用的工作状态，或者防止特定的机器学习陷阱（例如过度拟合）。

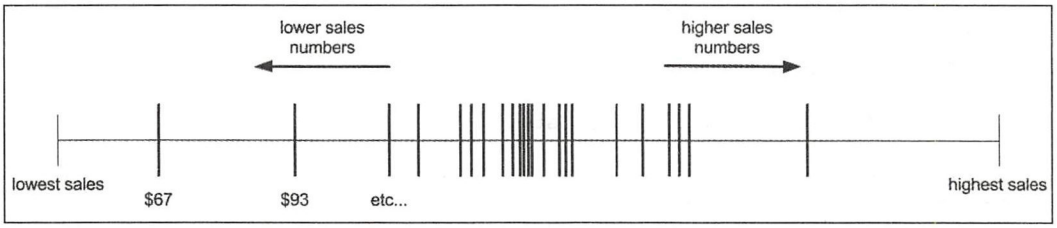
正如线性代数一样，在这里没法全面地介绍统计学的知识，但是有许多在线和出版资料可以学习统计学导论。本书将着重于基础概念的理解和利用 Go 的实践。这里将引入分布的概念，以及这些分布的量化和可视化的介绍。

2.2.1　分布

分布代表了一个数据集合中数值出现的频度。例如，作为数据科学家需要关注的一件事是，一个特定产品 / 服务的每日销售。假设已经有一个每日销售数据的很长列表（可以用向量或矩阵的一部分来表示）。这些销售数字是数据集合的一部分，其中包括一天的销售额是 121 美元，另一天的销售额是 207 美元，等等。

会有一个销售数字是所收集数据里面的最低值，同时也会有一个销售数字是所收集数据里面的最高值。剩下的销售数字是位于最高和最低之间的某个位置（至少当我们假设没有重复的时候）。下面的图形代表了在一条线段上的销售最低值、最高值、以及中间值：

仅供非商业用途或交流学习使用



这就是一个销售额的分布，至少是销售额分布的一个表现。请留意这个分布有一些区域的数字很多，而有一些区域数字很少。同时请留意这些数字从分布规律上看，似乎有靠近中间的趋势。

2.2.2 统计方法

本书会采用多种不同的统计方法来量化一个分布的样子。一般来说，这些方法分为两类：

1. **集中趋势测量法**：这些方法是测量大部分数值所在的位置，或者分布的中心（center）所在位置（例如，沿着之前直线所代表的）。
2. **离中趋势测量法**：这些方法是测量在分布范围内数值是如何分布的（从最低值到最高值）。

有各种各样的软件包可以快速计算或者利用这些方法。这里将使用 `gonum.org/v1/gonum/stat`（读者可能开始注意到我们将大量使用 `gonum`）和 `github.com/montanaflynn/stats`。

请留意，`gonum.org/v1/gonum/stat` 和 `github.com/montanaflynn/stats` 软件包的名字中有一个单词是不同的。在后面的例子中，请留意这种不同。

2.2.2.1 集中趋势测量法

集中趋势测量法包含以下几点

- **均值**：通常称为平均值。通过将分布中所有的数字求和，然后再除以数字的个数得到。
- **中位数**：如果将排列中的所有数字从最低到最高排列，这个数字就是划分最低一半和最高一半的数字。
- **众数**：在分布中出现次数最多的数字。

下面用前面介绍的第 1 章中提到的鸢尾花数据集中的一列来计算这些度量。提醒一下，这个数据集合包括 4 列花的测量值，其中一列是对应的花的种类。因此，每个测量列都包括一批数据，这些数据代表了该测量的分布。

仅供非商业用途或交流学习使用

```
// Open the CSV file.
irisFile, err := os.Open("../data/iris.csv")
if err != nil {
    log.Fatal(err)
}
defer irisFile.Close()

// Create a dataframe from the CSV file.
irisDF := dataframe.ReadCSV(irisFile)

// Get the float values from the "sepal_length" column as
// we will be looking at the measures for this variable.
sepalLength := irisDF.Col("sepal_length").Float()

// Calculate the Mean of the variable.
meanVal := stat.Mean(sepalLength, nil)

// Calculate the Mode of the variable.
modeVal, modeCount := stat.Mode(sepalLength, nil)

// Calculate the Median of the variable.
medianVal, err := stats.Median(sepalLength)
if err != nil {
    log.Fatal(err)
}

// Output the results to standard out.
fmt.Printf("\nSepal Length Summary Statistics:\n")
fmt.Printf("Mean value: %0.2f\n", meanVal)
fmt.Printf("Mode value: %0.2f\n", modeVal)
fmt.Printf("Mode count: %d\n", int(modeCount))
fmt.Printf("Median value: %0.2f\n\n", medianVal)
```

运行本程序会有如下输出：

```
$ go build
$ ./myprogram

Sepal Length Summary Statistics:
Mean value: 5.84
Mode value: 5.00
Mode count: 10
Median value: 5.80
```

可以看到均值、众数、中位数都有些不同。但是请注意在 sepal_length 列数值的平均值和中位数非常接近。

另一方面，如果将之前代码中的 sepal_length 更改为 petal_length，会得到如下的结果：

```
$ go build
$ ./myprogram

Sepal Length Summary Statistics:
Mean value: 3.76
Mode value: 1.50
Mode count: 14
Median value: 4.35
```

对于 petal_length 的数值，均值和中位数并不是特别相近。从这个信息中可以得到关于数据的一些直观感觉。如果均值和中位数并不接近，那说明高位或低位数值各自将平均值提高或降低了，而这个影响对中位数来说并不明显。这种情况称为**非正态分布**。

2.2.2.2 离中趋势测量法

现在已经知道大部分数值的分布位置（或者说分布的中心）。下面便来量化分布中的数值如何沿分布中心分布。以下是一些广泛使用的量化方法。

- **极大值**：分布中的最大数值
- **极小值**：分布中的最小数值
- **极差**：极大值与极小值之间的差异
- **方差**：这个方法是通过计算分布中的每个数值得到，计算分布中每个数值同均值之间的差值，将差值平方，然后同其他差值平方相加，接着除以分布中数值的数目。
- **标准差**：方差的平方根
- **分位数 / 四分位点**：同中位数类似，这些测量定义了分布中的临界点，使得一些低位数值在测量之下，而一些高位数值在测量之上。

使用 gonum.org/v1/gonum/stat，这些度量的计算如下所示：

```
// Open the CSV file.
irisFile, err := os.Open("../data/iris.csv")
if err != nil {
    log.Fatal(err)
}
defer irisFile.Close()

// Create a dataframe from the CSV file.
irisDF := dataframe.ReadCSV(irisFile)

// Get the float values from the "sepal_length" column as
// we will be looking at the measures for this variable.
sepalLength := irisDF.Col("petal_length").Float()

// Calculate the Max of the variable.
minVal := floats.Min(sepalLength)

// Calculate the Max of the variable.
maxVal := floats.Max(sepalLength)

// Calculate the Median of the variable.
rangeVal := maxVal - minVal

// Calculate the variance of the variable.
varianceVal := stat.Variance(sepalLength, nil)
```

```
// Calculate the standard deviation of the variable.
stdDevVal := stat.StdDev(sepalLength, nil)

// Sort the values.
inds := make([]int, len(sepalLength))
floats.Argsort(sepalLength, inds)

// Get the Quantiles.
quant25 := stat.Quantile(0.25, stat.Empirical, sepalLength, nil)
quant50 := stat.Quantile(0.50, stat.Empirical, sepalLength, nil)
quant75 := stat.Quantile(0.75, stat.Empirical, sepalLength, nil)

// Output the results to standard out.
fmt.Printf("\nSepal Length Summary Statistics:\n")
fmt.Printf("Max value: %0.2f\n", maxVal)
fmt.Printf("Min value: %0.2f\n", minVal)
fmt.Printf("Range value: %0.2f\n", rangeVal)
fmt.Printf("Variance value: %0.2f\n", varianceVal)
fmt.Printf("Std Dev value: %0.2f\n", stdDevVal)
fmt.Printf("25 Quantile: %0.2f\n", quant25)
fmt.Printf("50 Quantile: %0.2f\n", quant50)
fmt.Printf("75 Quantile: %0.2f\n\n", quant75)
```

运行程序将获得如下结果：

```
$ go build
$ ./myprogram

Sepal Length Summary Statistics:
Max value: 6.90
Min value: 1.00
Range value: 5.90
Variance value: 3.11
Std Dev value: 1.76
25 Quantile: 1.60
50 Quantile: 4.30
75 Quantile: 5.10
```

接下来处理完这些数字，然后看看在 sepal_length 列的分布数值中能发现什么。可以推导出如下结果。

首先，标准差是 1.76，数值的整体极差是 5.90。不同于方差，标准差与数值有着相同的计量单位。因此可以看到数值在取值范围内变化比较大（标准差的数值是整体极差的 30% 左右）。

其次，来看一看分位数，25% 分位数代表分布中的一个点，其中 25% 的数值分布在这个点之下，剩下的 75% 数值分布在这个点之上。50% 和 75% 分位数与其类似。25% 分位数同极小值之间的距离，小于 75% 分位数与极大值之间的距离。因此可以推导出一个结论，分布中的高位数值比低位数值更分散。

当然，可以在集中趋势测量法中利用这些测量的任何组合，以量化一个分布。并且还有一些其他统计方法没有在这里讲到。

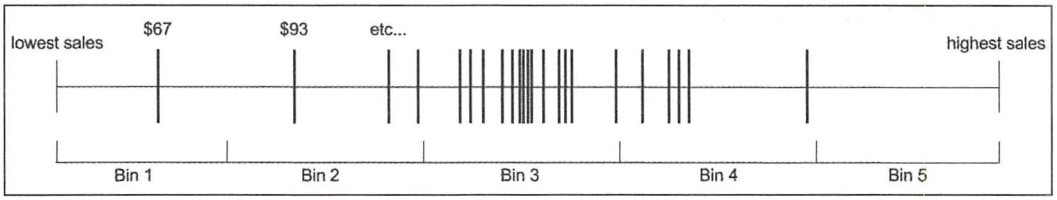
重要的是应当利用这些方法帮助建立数据的心理模型。这可以把结果连贯起来，并且并对工作进行完整性检查。

2.2.3 分布可视化

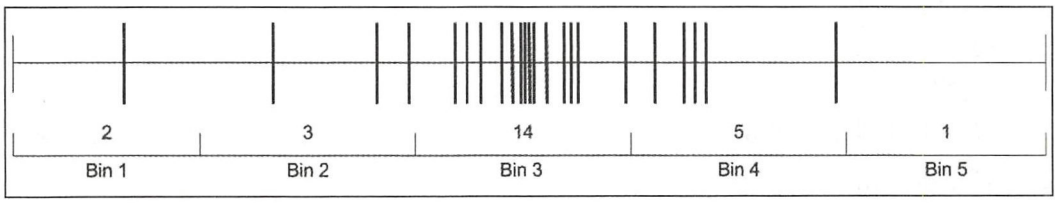
虽然量化一个分布很重要，但是可视化一个分布却是最直观的。有各种类型的图表和图形可以创建一个分布数值的可视化。这不仅有助于形成数据的心理模型，也有助于同团队成员之间、应用程序的用户之间等交流数据信息。

2.2.3.1 直方图

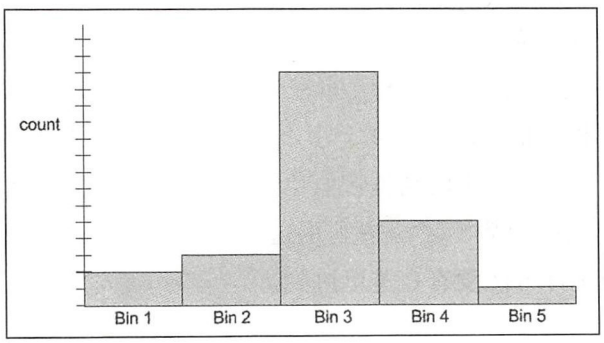
第一种帮助理解这些分布的图形 / 图表称为**直方图**。实际上，直方图是组织和计算数值的一个特定方法，这些数值可以在直方图中绘制。为了构造一个直方图，首先需要创建特定数目的箱子（bin），用以表示全部数值范围内的不同区域。以之前章节讨论的销售数据分布为例：



下一步，计算这些箱子里面每个有多少个数值。



这些计数，以及箱子的定义，形成我们的直方图。然后，我们可以很容易地将其转换为计数的图表，这为我们的分布提供了一个很好的可视化表示：



我们可以再次使用 gonum 从实际数据创建并绘制直方图。位于 gonum.org/v1/plot 的 gonum 软件包可以进行这种绘制，同时也支持其他类型的绘制。作为一个例子，我们为鸢尾花数据集中的每一列创建一个直方图。

首先从 gonum 导入以下内容：

```
import (
    "gonum.org/v1/plot"
    "gonum.org/v1/plot/plotter"
    "gonum.org/v1/plot/vg"
)
```

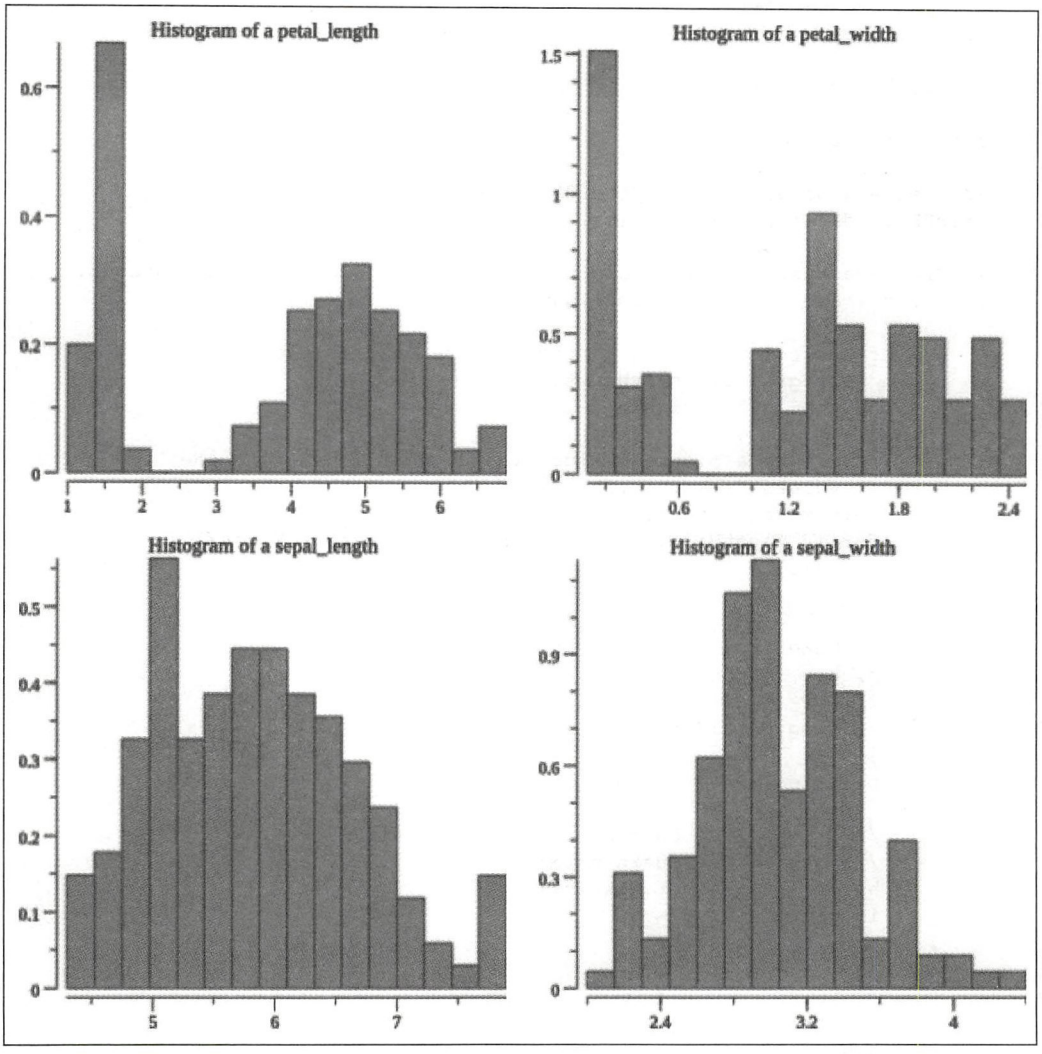
然后读取鸢尾花数据集，创建一个数据框（dataframe），然后查看由数字列生成的直方图：

```
// Open the CSV file.
irisFile, err := os.Open("../data/iris.csv")
if err != nil {
    log.Fatal(err)
}
defer irisFile.Close()

// Create a dataframe from the CSV file.
irisDF := dataframe.ReadCSV(irisFile)

// Create a histogram for each of the feature columns in the dataset.
for _, colName := range irisDF.Names() {

    // If the column is one of the feature columns, let's
    // a histogram of the values.
    if colName != "species" {

        // Create a plotter.Values value and fill it with the
        // values from the respective column of the dataframe.
        v := make(plotter.Values, irisDF.Nrow())
        for i, floatVal := range irisDF.Col(colName).Float() {
            v[i] = floatVal
        }

        // Make a plot and set its title.
        p, err := plot.New()
        if err != nil {
            log.Fatal(err)
        }
        p.Title.Text = fmt.Sprintf("Histogram of a %s", colName)

        // Create a histogram of our values drawn
        // from the standard normal.
        h, err := plotter.NewHist(v, 16)
        if err != nil {
            log.Fatal(err)
        }

        // Normalize the histogram.
        h.Normalize(1)
```

```
            // Add the histogram to the plot.
            p.Add(h)

            // Save the plot to a PNG file.
            if err := p.Save(4*vg.Inch, 4*vg.Inch, colName+"_hist.png"); err !=
nil {
                log.Fatal(err)
            }
        }
    }
```

请注意，这里已经规范了直方图（使用 h.Normalize() 方法）。这是常用的做法，因为通常会需要对比不同数值个数的分布之间的差异。规范化的直方图可以并排比较不同的分布。

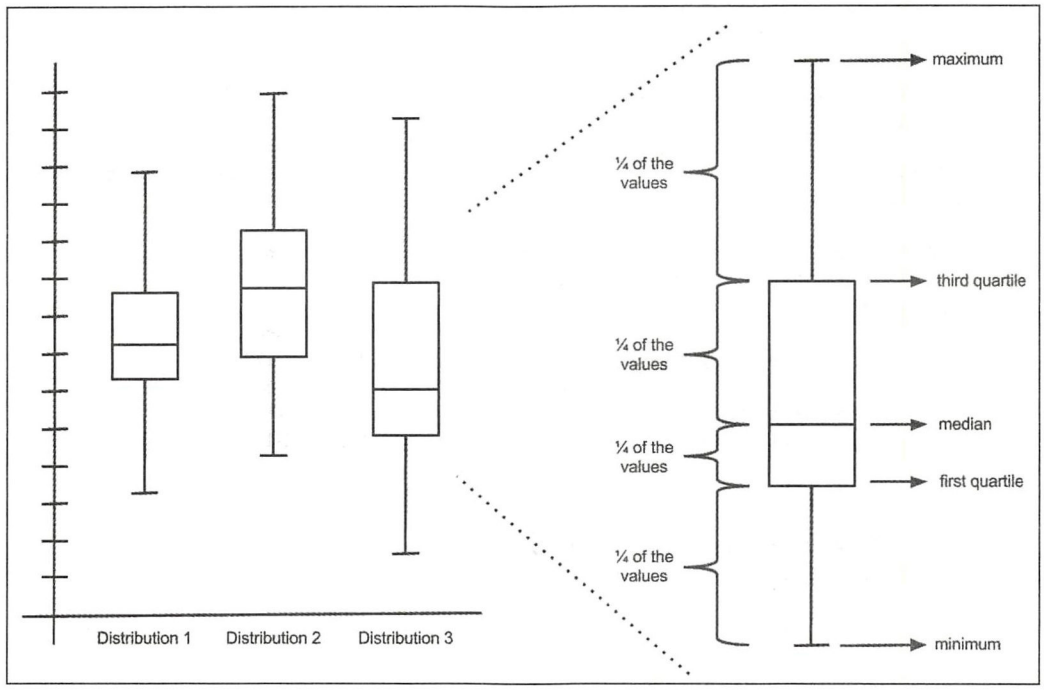
如下所示的直方图，前面的代码将为鸢尾花数据集中的数字列生成四个 *.png 文件：



这些分布中的每一个看起来都不同。sepal_width 分布看起来像钟形曲线或正态 / 高斯分布（这会在本书后面讨论）。另一方面，pedal_width 的分布看起来有两种明显不同的数值块，稍后将在开发机器学习工作流程时应用这些不同的发现，但现在只要注意这些可视化如何帮助开发数据的心理模型。

2.2.3.2　箱形图

直方图绝不是视觉上了解数据的唯一方法，另一种常用的图表类型称为**箱形图**。这种类型的图表同样也能提供分布中的数据分组和离散情况，但与直方图相比，箱形图有几个明显的特征可以帮助理解数据：



因为箱形图中的边框的边界由中位数，**第一四分位数**（first quartile，25% 分位数 / 百分位数）和**第三四分位数**（third quartile）定义，所以两个中央框中包含相同数量的分布值。如果一个箱子比另一个箱子大，则意味着是偏态分布。

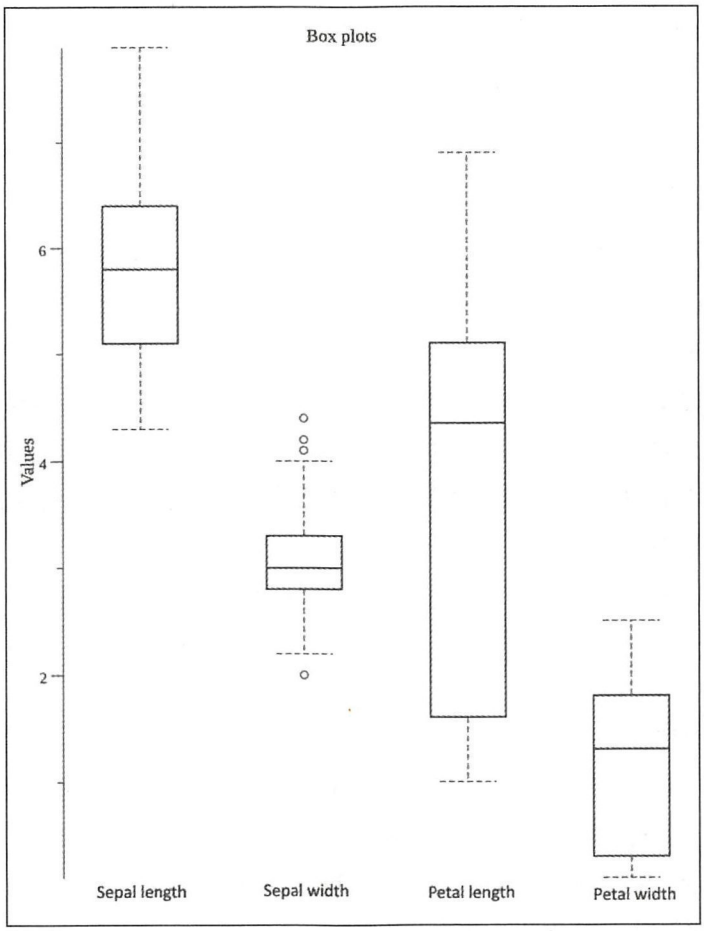
箱形图中还有两条尾巴或胡须，与包含大部分数据的区域（中间的 50%）相比，这些数据可以快速显示分布范围。

为了巩固对这种箱形图的理解，让我们再次创建鸢尾花数据集的的箱形图。类似于直方图，使用 gonum.org/v1/plot。但是这时候，会把所有的箱形图绘制在同一个 *.png 文件中：

```
// Open the CSV file.
irisFile, err := os.Open("../data/iris.csv")
if err != nil {
    log.Fatal(err)
}
defer irisFile.Close()

// Create a dataframe from the CSV file.
irisDF := dataframe.ReadCSV(irisFile)

// Create the plot and set its title and axis label.
p, err := plot.New()
if err != nil {
    log.Fatal(err)
}

p.Title.Text = "Box plots"
p.Y.Label.Text = "Values"

// Create the box for our data.
w := vg.Points(50)

// Create a box plot for each of the feature columns in the dataset.
for idx, colName := range irisDF.Names() {

    // If the column is one of the feature columns, let's create
    // a histogram of the values.
    if colName != "species" {

        // Create a plotter.Values value and fill it with the
        // values from the respective column of the dataframe.
        v := make(plotter.Values, irisDF.Nrow())
        for i, floatVal := range irisDF.Col(colName).Float() {
            v[i] = floatVal
        }

        // Add the data to the plot.
        b, err := plotter.NewBoxPlot(w, float64(idx), v)
        if err != nil {
            log.Fatal(err)
        }
        p.Add(b)
    }
}

// Set the X axis of the plot to nominal with
// the given names for x=0, x=1, etc.
p.NominalX("sepal_length", "sepal_width", "petal_length", "petal_width")

if err := p.Save(6*vg.Inch, 8*vg.Inch, "boxplots.png"); err != nil {
    log.Fatal(err)
}
```

这会生成如下图表：

仅供非商业用途或交流学习使用



正如在直方图中观察到的那样，sepal_length 列显得相对而言更加对称。另一方面，petal_length 显得更加不对称。注意，这说明 gonum 已经包含了几个离群值（标记为圆圈或圆点）。许多绘图软件包都包括这种功能。这表明这些数值离分布的中位数至少还有一定的距离。

2.3　概率论

目前，已经介绍了几种表示 / 操纵数据（矩阵和向量）的方法，并且知道了如何获得和理解数据，以及如何优化数据的外观（统计数据）。然而在开发机器学习的应用时，有时还想知道在给定历史结果的情况下，预测准确的可能性，或者特定结果的重要性。概率可以帮助我们回答这些问题的可能性（how likely）和重要性（how significant）。

一般来说，概率与事件或观察的可能性有关。例如，如果抛硬币做出决定，有

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

多大可能会看到正面（50%）或反面（50%），或者说该硬币是公平硬币的可能性是多少？这可能看起来是一个微不足道的例子，但进行机器学习时，类似问题就会出现。实际上，一些机器学习算法是建立在概率的规则和定理之上的。

2.3.1 随机变量

比方说，有一个实验，就像翻转硬币的情景一样，可能有多个结果（正面或反面）。现在来定义一个变量，其值可能是这些结果之一。这个变量被称为**随机变量**（random variable）。

在抛硬币的情况下（至少就公平硬币而言），随机变量的每个结果是具有相同的可能性。也就是说，有 50% 的机会看到正面的结果，50% 的机会看到反面的结果。然而随机变量的各种取值无须相等。如果试图预测是否会下雨，那么这些结果就不会有同样的可能性。

随机变量使我们能够定义这些问题的可能性和重要性。它们可以具有有限数量的结果和 / 或可以表示连续变量的范围。

2.3.2 概率测量

在一个实验中观察到某种特定结果的可能性有多大？这里引入**概率测量**（probability measure）用以定量地回答这个问题，通常也被称为概率。它们由 0 和 1 之间的数字表示，和 / 或以 0% 和 100% 之间的百分比表示。

在翻转公平硬币的情况下，有以下情景：

- 有 0.5 或 50% 的概率是正面，这里的 0.5 或 50% 是概率测量值
- 有 0.5 或 50% 的可能性是反面

某个实验的概率测量的和值必须为 1，因为事件发生时，它必须对应于其中一个可能的结果。

2.3.3 独立和条件概率

如果一个事件发生的概率不影响其他事件的概率，那么这两个事件（实验的结果）就是独立的。独立事件的一个例子是硬币的翻转或骰子的滚动。另一方面，依赖事件是一个事件的概率影响另一个事件的概率的事件。依赖性事件的一个例子是从一副牌中抽出牌而不更换。

第二种概率通常被称为条件概率。条件概率如何量化呢？独立的概率可以用符号 $P(\mathrm{A})$ 表示，这是 A 的概率（其中 A 可以表示翻转硬币，掷骰子等）。那么条件概率由 $P(B \mid A)$ 表示，它是指在 B（其中 B 是另一个概率事件）发生的条件下，A 的概率。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

为了计算条件概率，我们可以使用贝叶斯定理 / 规则：$P(A \mid B)=P(B \mid A)\ P(A)/P(B)$。有时你会看到这些术语定义如下：

- $P(A \mid B)$：后验概率，这是在观察到 B 发生之后 A 的概率
- $P(A)$：先验概率，这是在 B 发生之前的，关于 A 的概率
- $P(B \mid A)$：似然概率，因为它是 B 与 A 的兼容性的量度
- $P(B)$：实证概率，因为这是测量 B 发生的概率，已知该事件已经发生

这个定理是本书后面将讨论的各种技术的基础，比如朴素贝叶斯分类技术。

2.3.4　假设检验

可以用概率量化可能性问题的概率，甚至可以用贝叶斯定理计算条件概率，但是如何量化真实世界的观测结果的重要性呢？例如，可以用一枚公平硬币来量化正面 / 反面的概率，但是假如抛掷一枚硬币多次，结果是 48% 的头像和 52% 的反面，如何判断这种结果的重要性？它是重要的吗？这是否意味着使用了一枚非公平硬币？

这些重要性问题可以使用被称为假设检验的方法来回答。该方法一般包括以下步骤：

1. 设定一个**零假设**，称为 H_0，另一个**对立假设**称为 Ha。H_0 代表你观察到的情况（例如，48% 的头像和 52% 的反面）是纯粹随机的结果，而 Ha 代表某种类型的潜在影响导致明显偏离随机的情况（例如，不均匀的硬币）。零假设总是假定为真。
2. 设定一个**检验统计量**，用来确定 H_0 的有效性。
3. 设定一个 **p 值**，它表示观察一个检验统计量的概率，这个概率至少和假设 H_0 为真的检验统计量一样显著。这个 p 值可以从对应于检验统计量的概率分布（通常表示为表格或分布函数）中获得。
4. 将获取的 p 值与预定的阈值进行比较。如果 p 值小于或等于预定阈值，则排除 H_0 并使用 Ha。

这看起来可能有些抽象，但是这个方法会在某些时候与机器学习工作流程相关联。比如，更改某个机器学习模型用以优化广告投放，然后可能需要量化销售增长是否具有统计显著性。在另一种情况下，可以对欺诈性网络流量的日志分析有所帮助，并且可以借此构建一个模型，用以识别与预期网络流量的统计的显著偏差。

请注意，你也可能会看到一些称为 A/B 测试的假设检验，虽然这里列出的方法是常见的，但它绝不是假设检验的唯一方法。例如还有贝叶斯 A/B 测试的方法，优化的劫持算法等，本书不会详细介绍。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

2.3.4.1 测试统计

在假设检验中可以使用许多不同的检验统计量。这些包括 Z 统计量、T 统计量、F 统计量和卡方统计量。当然，可以在 Go 中从头开始实现这些统计算法，这并不会有太多的麻烦。但是，也可以使用一些预设。

回到 gonum.org/v1/gonum/stat，可以用如下所示方法计算卡方统计量：

```
// Define observed and expected values. Most
// of the time these will come from your
// data (website visits, etc.).
observed := []float64{48, 52}
expected := []float64{50, 50}

// Calculate the ChiSquare test statistic.
chiSquare := stat.ChiSquare(observed, expected)
```

2.3.4.2 计算 p 值

假设有以下情景：

对某地居民的调查显示，所有居民中有 60% 没有经常锻炼，25% 是零星锻炼，15% 经常锻炼。在做了一些建模并提供一些社区服务之后，调查重复了相同的问题。跟踪调查由 500 名居民完成，结果如下：

没有经常锻炼：260

零星锻炼：135

经常锻炼：105

总数：500

现在，要确定是否有证据表明居民的反应在统计上有重大的转变。零假设和对立假设如下：

- H_0：与先前观察到的百分比的偏差是由于纯粹的随机事件
- Ha：这种偏差是由于纯粹随机事件之外的一些潜在影响（可能是新社区服务）

首先，使用卡方检验统计量来计算检验统计量：

```
// Define the observed frequencies.
observed := []float64{
    260.0, // This number is the number of observed with no regular
exercise.
    135.0, // This number is the number of observed with sporatic exercise.
    105.0, // This number is the number of observed with regular exercise.
}
// Define the total observed.
totalObserved := 500.0
// Calculate the expected frequencies (again assuming the null Hypothesis).
expected := []float64{
    totalObserved * 0.60,
    totalObserved * 0.25,
    totalObserved * 0.15,
}
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
// Calculate the ChiSquare test statistic.
chiSquare := stat.ChiSquare(observed, expected)

// Output the test statistic to standard out.
fmt.Printf("\nChi-square: %0.2f\n", chiSquare)
```

这会得到如下卡方值：

```
$ go build
$ ./myprogram

Chi-square: 18.13
```

接下来，需要计算这个卡方对应的 p 值。这要求我们知道关于**卡方分布**的信息，卡方分布定义了某些卡方度量以及**自由度**的 p 值。github.com/gonum/stat 中也包含这个卡方分布的表示，可以从中计算出 p 值：

```
// Create a Chi-squared distribution with K degrees of freedom.
// In this case we have K=3-1=2, because the degrees of freedom
// for a Chi-squared distribution is the number of possible
// categories minus one.
chiDist := distuv.ChiSquared{
    K: 2.0,
    Src: nil,
}

// Calculate the p-value for our specific test statistic.
pValue := chiDist.Prob(chiSquare)

// Output the p-value to standard out.
fmt.Printf("p-value: %0.4f\n\n", pValue)
```

这给出了如下结果：

```
$ go build
$ ./myprogram

Chi-square: 18.13
p-value: 0.0001
```

因此，第 2 版调查结果中看到的结果有 0.01% 的可能性是纯粹偶然的。例如，如果使用 5% 的阈值（这是常用的一个阀值），将需要拒绝零假设并采用替代假设。

2.4　参考书目

向量和矩阵：

- gonum.org/v1/gonum/floats 文档：https://godoc.org/gonum.org/v1/gonum/floats
- gonum.org/v1/gonum/mat 文档：https://godoc.org/gonum.org/v1/gonum/mat

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

统计：

- gonum.org/v1/gonum/stat 文档：https://godoc.org/gonum.org/v1/gonum/stat
- github.com/montanaflynn/stats 文档：https://godoc.org/github.com/montanaflynn/stats

可视化：

- gonum.org/v1/plot 文档：https://godoc.org/gonum.org/v1/plot
- gonum.org/v1/plot wiki 和例子：https://github.com/gonum/plot/wiki/Example-plots

可能性：

- gonum.org/v1/gonum/stat/distuv 文档：https://godoc.org/gonum.org/v1/gonum/stat/distuv

2.5 小结

前面介绍的 Go 中矩阵、线性代数、统计学以及概率为我们提供了一整套工具，用以理解、构造和操作数据。这套工具将应用到整本书中，用来处理各种各样的机器学习问题，甚至还可以在机器学习之外的环境中使用。然而，下一章中，本书将讨论一些在机器学习环境中非常重要的思想和技术，它们是评估和验证。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

第 3 章
评估和验证

为了获得持续、可靠的机器学习流程，并开发出真正有价值的机器学习应用，需要能够测量机器学习模型的性能，同时也需要确保机器学习模型能够推广到实际生产中看到的数据。否则就是在毫无根据地尝试。这会导致不了解模型的预期行为，以及无法随时间逐步改进模型。

测量模型如何执行（关于特定数据）的过程称为**评估**（evaluation）。确保模型能推广到可能遇到的数据的过程称为**验证**。这两个过程需要运用到每个机器学习流程和应用中，本章将介绍这两个过程。

3.1 评估

科学的基本原则便是测量，机器学习科学也不例外。需要能够测量、评估模型的性能如何，以便持续提高模型的质量、对比模型间的差异、检测模型何时性能不佳。

这里仅有一个问题，如何评估模型的性能？应当测量模型的训练速度或者做出推论的速度吗？应当测量得到正确答案的次数吗？如何得知正确答案是什么呢？应当测量偏离观测值的距离吗？如何测量偏离距离？

如你所见，围绕如何评估模型，需要做许多决策。真正重要的是机器学习的应用场景。在一些情况下，效率是至关重要的，但是在每个机器学习环境中，均要求测量预测、推论或者结果是否与理想的预测、推论以及结果匹配。因此计算结果与理想结果的比对应当优先于速度优化。

一般来说，有如下几种类型的结果需要评估。

- **连续的**：例如总销售额、股票价格、温度这些有连续数值分布的结果

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

（12 102.21 美元，92 度等）

- **分类的**：例如欺诈 / 非欺诈、活动、能被有限数字分类的名字等结果（欺诈、站立、Frank 等）

上述类型的每种结果都有相应的评估指标，这些指标本书均会论述。然而，请记住评估指标的选择取决于使用机器学习模型想要达到的目标。并没有全能的指标，在某些情况下，可能要创建自己的评估指标。

3.1.1 连续指标

假设有一个模型用于预测一些连续的数值，例如股票价格。同时假设已经积累了一些可以同实际观察值进行比较的预测值：

```
observation,prediction
22.1,17.9
10.4,9.1
9.3,7.8
18.5,14.2
12.9,15.6
7.2,7.4
11.8,9.7
...
```

现在的问题是，如何测量模型的性能？第一步是计算观察值和预测值之间的不同，以获得一个误差（error）：

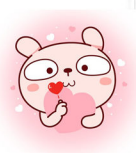
```
observation,prediction,error
22.1,17.9,4.2
10.4,9.1,1.3
9.3,7.8,1.5
18.5,14.2,4.3
12.9,15.6,-2.7
7.2,7.4,-0.2
11.8,9.7,2.1
...
```

通过这个误差可以大致了解，当前预测离实际的值有多远。然而，单独查看所有的误差值并不实际，特别是当数据量很庞大的时候。可能有上百万个误差值。因此需要一个方法在总体上理解误差。

均方误差（MSE）和**平均绝对误差**（MAE）提供了关于总体误差的视角。

- MSE 或者**均方误差**（MSD）是所有误差的平方的平均值
- MAE 是所有误差绝对值的平均值

MES 和 MAE 都提供了非常好的预测准确度的总体视角，但侧重点不同。MSE 会将误差平方，所以会比 MAE 更夸大了大的误差值（例如，异常值的情况）。换句



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

话说 MSE 对异常值更敏感。另一方面，MAE 将尝试预测的变量维持在同一单位，因此可直接将这些值进行比较。

对于这个数据集，可以解析观测值和预测值，并按以下方式计算 MAE 和 MSE：

```
// Open the continuous observations and predictions.
f, err := os.Open("continuous_data.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a new CSV reader reading from the opened file.
reader := csv.NewReader(f)

// observed and predicted will hold the parsed observed and predicted
values

// form the continuous data file.
var observed []float64
var predicted []float64

// line will track row numbers for logging.
line := 1
// Read in the records looking for unexpected types in the columns.
for {

    // Read in a row. Check if we are at the end of the file.
    record, err := reader.Read()
    if err == io.EOF {
        break
    }

    // Skip the header.
    if line == 1 {
        line++
        continue
    }

    // Read in the observed and predicted values.
    observedVal, err := strconv.ParseFloat(record[0], 64)
    if err != nil {
        log.Printf("Parsing line %d failed, unexpected type\n", line)
        continue
    }

    predictedVal, err := strconv.ParseFloat(record[1], 64)
    if err != nil {
        log.Printf("Parsing line %d failed, unexpected type\n", line)
        continue
    }

    // Append the record to our slice, if it has the expected type.
    observed = append(observed, observedVal)
    predicted = append(predicted, predictedVal)
    line++
}
```

仅供非商业用途或交流学习使用

```
// Calculate the mean absolute error and mean squared error.
var mAE float64
var mSE float64
for idx, oVal := range observed {
    mAE += math.Abs(oVal-predicted[idx]) / float64(len(observed))
    mSE += math.Pow(oVal-predicted[idx], 2) / float64(len(observed))
}

// Output the MAE and MSE value to standard out.
fmt.Printf("\nMAE = %0.2f\n", mAE)
fmt.Printf("\nMSE = %0.2f\n\n", mSE)
```

对于示例数据，结果如下：

```
$ go build
$ ./myprogram

MAE = 2.55

MSE = 10.51
```

为了评价这些数值的好坏，需要同实际观测数据进行对比。在这里，MAE 是 2.55，观测的平均值是 14.0。因此 MAE 是平均值的大约 20%。从上下文来看，并不是很好。

同 MSE 和 MAE 一块，还可能看到 R 方（R-squared）（又称 R^2 或者 R2），或者称为**决定系数**（coefficient of determination），也能作为连续变量模型的评估指标。虽然 R 方也可以给出预测偏差的趋势，但是它们的理念有一些不同。

R 方测量的是，预测值中所捕获的观察值的方差所占比例。请留意尝试预测的数值有一些可变性。例如可能尝试预测股票价格、利率或者疫情进展，而这些在本质上并不完全相同。这里尝试创建一个模型以预测观察值中的此类变化，所捕获的变化的百分比则用 R 方表示。

gonum.org/v1/gonum/stat 有内置的函数可以方便地计算 R 方：

```
// Calculate the R^2 value.
rSquared := stat.RSquaredFrom(observed, predicted, nil)

// Output the R^2 value to standard out.
fmt.Printf("\nR^2 = %0.2f\n\n", rSquared)
```

对示例数据集运行上述代码，可以得到如下结果：

```
$ go build
$ ./myprogram

R^2 = 0.37
```

这个 R 方是好是坏呢？请记住 R 方是一个百分比，并且越高越好。这里在尝试预测的变量中捕获到 37% 的变化。这不是很好。

3.1.2　分类指标

假设已经有一个模型用于预测一些离散数值，例如欺诈 / 非欺诈，站立 / 坐着 / 行走，已批准 / 未批准，等等。数据看起来可能如下所示：

```
observed,predicted
0,0
0,1
2,2
1,1
1,1
0,0
2,0
0,0
...
```

观测值可能是任何有限数值之一（在本例中为 1、2 或者 3），每个取值代表数据的离散类别之一（例如类别 1 可能表示欺诈交易，类别 2 可能表示非欺诈交易，类别 3 可能表示无效的交易）。预测值也可以采用这些离散数值之一。在评估预测的时候，需要一些方法来衡量这些离散预测的正确性。

3.1.2.1　分类变量的个体评估指标

实际上，有很多方法可用来评估离散预测，其中包括准确度、精确度、召回率、特异性、灵敏度、缺陷、错误遗漏率等。和连续变量一样，并没有万能的评估指标。每次研究一个问题的时候，都需要确定适合问题的指标，并与项目目标相匹配。不要去尝试优化不合适的指标，那样只会浪费大量时间去重新实现一个基于其他指标的模型。

为了理解这些指标，并决定哪种更适合自己，需要认识到在做离散预测的时候，可能会出现一些不同的情况。

- **真阳性**（True Positive，TP）：预测某个类别，而观察到的正是该类别（例如，预测的是欺诈，观察到的也是欺诈）
- **假阳性**（False Positive，FP）：预测某个类别，而观察到的却是另一个类别（例如，预测是欺诈，然而观察到的是非欺诈）
- **真阴性** (True Negative，TN)：预测观察值不会是某个类别，实际观察值也不是那个类别（例如，预测不是欺诈，实际上也不是欺诈）
- **假阴性**（False Negative，FN）：预测观察值不是某个类别，然而实际上观察值又是该类别（例如，预测不是欺诈，但是观察到的是欺诈）

可以看到有许多方法可以组合、聚合和测量这些场景。实际上，针对当前特定问题，可以采用独特的方法来聚合 / 测量它们。然而，有很多标准的方法可以聚合和测量这些场景，并生成如下常见的指标。

- **准确度**（Accuracy）：预测正确的百分比，或 $(TP+TN)/(TP+TN+FP+FN)$
- **精确度**（Precision）：阳性预测中实际阳性的比率，或 $TP/(TP+FP)$
- **召回率**（Recall）：阳性预测中被标注为阳性的比率，或 $TP/(TP+FN)$

即便我在这里强调这些指标，读者也应该看看其他通用指标及其含义。在这里可以了解相关概念：https://en.wikipedia.org/wiki/Precision_and_recall。

如下是解析数据和计算准确度的例子，首先读取 `labeled.csv` 文件，创建 CSV reader，然后初始化两个切片以保存解析后的观测数据和预测数据。

```
// Open the binary observations and predictions.
f, err := os.Open("labeled.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a new CSV reader reading from the opened file.
reader := csv.NewReader(f)

// observed and predicted will hold the parsed observed and predicted
values
// form the labeled data file.
var observed []int
var predicted []int
```

然后遍历 CSV 中的记录并解析其数值，对比观测值和预测值以计算准确度：

```
// line will track row numbers for logging.
line := 1

// Read in the records looking for unexpected types in the columns.
for {

    // Read in a row. Check if we are at the end of the file.
    record, err := reader.Read()
    if err == io.EOF {
        break
    }

    // Skip the header.
    if line == 1 {
        line++
        continue
    }

    // Read in the observed and predicted values.
    observedVal, err := strconv.Atoi(record[0])
    if err != nil {
        log.Printf("Parsing line %d failed, unexpected type\n", line)
        continue
    }
```

```
    predictedVal, err := strconv.Atoi(record[1])
    if err != nil {
        log.Printf("Parsing line %d failed, unexpected type\n", line)
        continue
    }

    // Append the record to our slice, if it has the expected type.
    observed = append(observed, observedVal)
        predicted = append(predicted, predictedVal)
        line++
    }

    // This variable will hold our count of true positive and
    // true negative values.
    var truePosNeg int

    // Accumulate the true positive/negative count.
    for idx, oVal := range observed {
        if oVal == predicted[idx] {
            truePosNeg++
        }
    }

    // Calculate the accuracy (subset accuracy).
    accuracy := float64(truePosNeg) / float64(len(observed))

    // Output the Accuracy value to standard out.
    fmt.Printf("\nAccuracy = %0.2f\n\n", accuracy)
```

运行以上代码会得到以下结果：

```
$ go build
$ ./myprogram

Accuracy = 0.97
```

结果是 97%！非常不错，这意味着本次的准确度是 97%。

同样可以计算出精确度和召回率。但是你可能已经注意到，当有两个以上类别或种类时，可以用多种方法来实现。可以将第 1 类当作肯定，其他类别当作否定；或者将第 2 类当作肯定，其他类别当作否定，等等。也就是说，可以为每个类别计算精确度和召回率。示例代码如下所示：

```
    // classes contains the three possible classes in the labeled data.
    classes := []int{0, 1, 2}

    // Loop over each class.
    for _, class := range classes {

        // These variables will hold our count of true positives and
        // our count of false positives.
        var truePos int
        var falsePos int
        var falseNeg int
```

```
    // Accumulate the true positive and false positive counts.
    for idx, oVal := range observed {

        switch oVal {

        // If the observed value is the relevant class, we should check to
        // see if we predicted that class.
        case class:
            if predicted[idx] == class {
                truePos++
                continue
            }

            falseNeg++

        // If the observed value is a different class, we should
        // check to see if we predicted a false positive.
        default:
            if predicted[idx] == class {
                falsePos++
            }
        }
    }

    // Calculate the precision.
    precision := float64(truePos) / float64(truePos+falsePos)

    // Calculate the recall.
    recall := float64(truePos) / float64(truePos+falseNeg)

    // Output the precision value to standard out.
    fmt.Printf("\nPrecision (class %d) = %0.2f", class, precision)
    fmt.Printf("\nRecall (class %d) = %0.2f\n\n", class, recall)
}
```

运行代码，结果如下所示：

```
$ go build
$ ./myprogram

Precision (class 0) = 1.00
Recall (class 0) = 1.00


Precision (class 1) = 0.96
Recall (class 1) = 0.94


Precision (class 2) = 0.94
Recall (class 2) = 0.96
```

请注意精确度和召回率是两个有细微差异的指标，并且有不同的含义。如果希望得到整体的精确度和召回率，应将每个类别的精确度和召回率取平均值。实际上，如果某个类别比其他类别更加重要，应该采用加权平均数计算，并将其作为评估指标。

当得到一些指标为 100% 时，看起来结果不错，但实际上可能隐藏着问题，本书会进一步探讨这一点。

某些情况下，例如金融或者银行业务，一些类别的假阳性或者其他情况代价会很高。例如，将交易错误地标记为欺诈行为可能导致重大损失。另一方面，其他类别的某些结果有可能可以忽略不计。这些场景可能需要使用自定义的度量或者价值函数，以确保特定种类、特定结果或者特定结果的组合的权重高于其他的。

3.1.2.2　混淆矩阵、曲线下面积和接受者操作特征

除了计算模型的单个数值指标外，还有许多技术可以将不同指标组合成一种形式，以更完整地展示模型性能。它们包括但不局限于：**曲线下面积**（AUC）和**接受者操作特征**（ROC）**曲线**。

混淆矩阵能够可视化以二维形式预测各种 TP、TN、FP 和 FN 数值。一个混淆矩阵的行对应期望预测的种类，列对应实际预测的种类。然后每个元素的值是对应的数量：

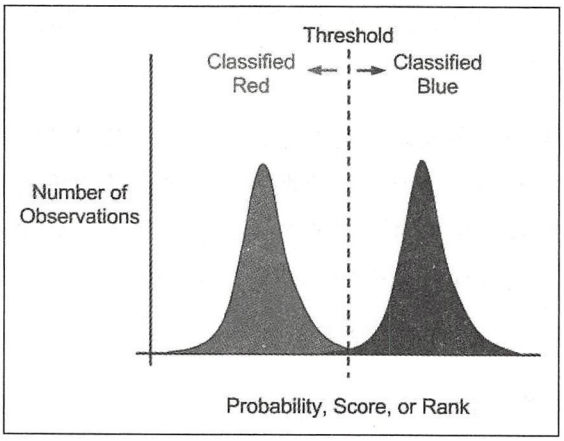
		Predicted	
		Fraud	Not Fraud
Observed	Fraud	TP	FN
	Not Fraud	FP	TN

如上所见，理想情况是混淆矩阵只有对角线的条目（TP，TN）。对角线元素表示预测一个特定种类，观察值也正是那个种类。非对角线元素表示错误预测的数量。

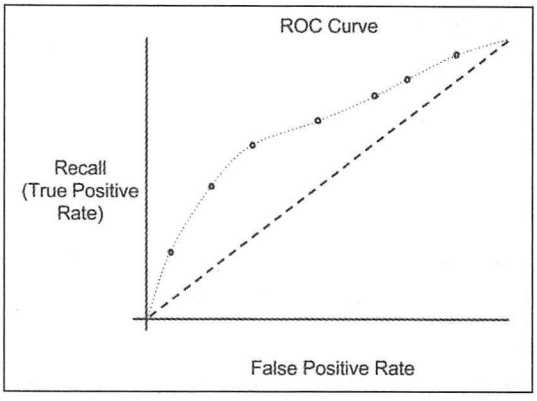
这种混淆矩阵对超过两个类别的问题会特别有帮助。例如，可能会尝试预测基于移动加速器和位置数据的各种活动。这些活动可能包括两个以上的种类，例如站立、坐下、跑步、驾驶等。这个问题的混淆矩阵规模会超过 2×2，以此可以快速评估模型在所有类别中的整体性能，并且识别模型中性能不佳的类别。

除混淆矩阵之外，ROC 曲线也经常用作全面了解二元分类器（或者被训练从两个分类中预测其中一个的模型）的性能。ROC 曲线针对所有可能的分类阈值，标识了召回率和假阳性比率（$FP/(FP+TN)$）。

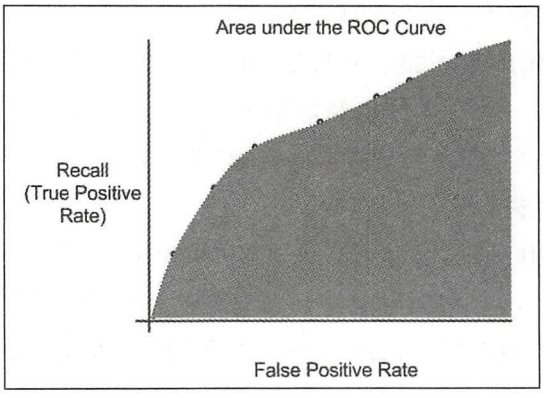
ROC 曲线中使用的阈值代表用于划分不同分类的边界或者等级。也就是说，由 ROC 曲线评估的模型必须根据可能性、等级或者分数（在下图中称为分数）对两个类别进行预测。在之前提到的所有样本中，分数都是单项分类的，反之亦然：



在测试示例中（召回率，假阳性比率），依次用点表示每个分数或者等级，然后将这些点连接起来形成一条曲线，这条曲线就是 ROC 曲线。很多情况下会在 ROC 曲线的对角线上绘制一条直线，这条直线是分类器的参考线，具有近似随机的预测能力：



一个好的 ROC 曲线是在图的左上部分，这意味着所采用的模型比随机预测能力更好。ROC 曲线越接近图表的左上角越好。这意味着良好的 ROC 曲线具有更多的 AUC，ROC 曲线的 AUC 也用作评估指标。参考下图：



仅供非商业用途或交流学习使用

gonum.org/v1/gonum/stat 有一些内置函数和类型可帮助构建 ROC 曲线和 AUC 指标：

```
func ROC(n int, y []float64, classes []bool, weights []float64) (tpr, fpr
[]float64)
```

以下是一个快速示例，可用于计算 ROC 曲线的 AUC：

```
// Define our scores and classes.
scores := []float64{0.1, 0.35, 0.4, 0.8}
classes := []bool{true, false, true, false}

// Calculate the true positive rates (recalls) and
// false positive rates.
tpr, fpr := stat.ROC(0, scores, classes, nil)
// Compute the Area Under Curve.
auc := integrate.Trapezoidal(fpr, tpr)

// Output the results to standard out.
fmt.Printf("true positive rate: %v\n", tpr)
fmt.Printf("false positive rate: %v\n", fpr)
fmt.Printf("auc: %v\n", auc)
```

运行此代码可以得到如下结果：

```
$ go build
$ ./myprogram
true positive rate: [0 0.5 0.5 1 1]
false positive rate: [0 0 0.5 0.5 1]
auc: 0.75
```

3.2　验证

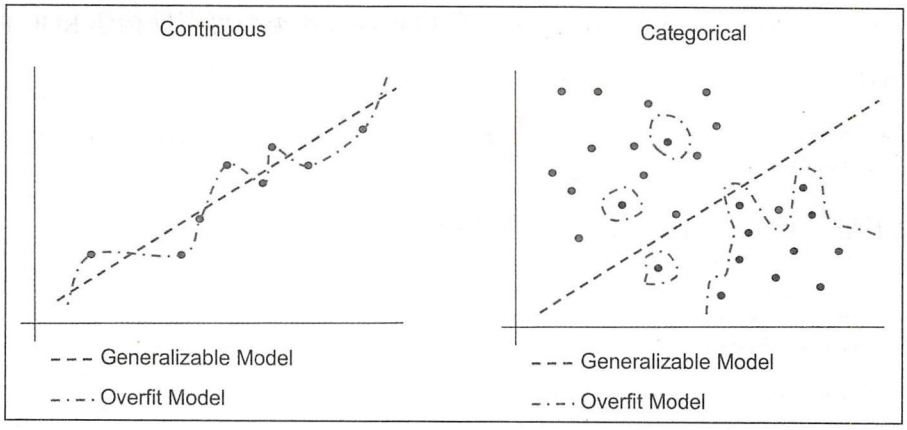
现在已经介绍了一些衡量模型表现如何的方法。事实上，可以创建一个超级复杂的模型，用以预测每一个观察结果而不会出错。例如，可以创建一个模型，该模型采用观察行的索引并返回每个行的确切答案。它可能是有很多参数的、非常庞大的函数，但它会返回正确的答案。

那么这样做有什么问题呢？问题是它无法推广到新数据。复杂的模型可以很好地预测已经使用的数据，但是一旦用这个模型验证一些新的输入数据（新数据不是训练数据集的一部分），该模型可能表现不佳。

我们称这种类型的（无法推广的）模型为**过度拟合**（overfit）的模型。也就是说，根据可用的数据，使模型变得越来越复杂的过程就是过度拟合模型。

预测连续值或离散 / 分类值时可能会发生过度拟合：

仅供非商业用途或交流学习使用



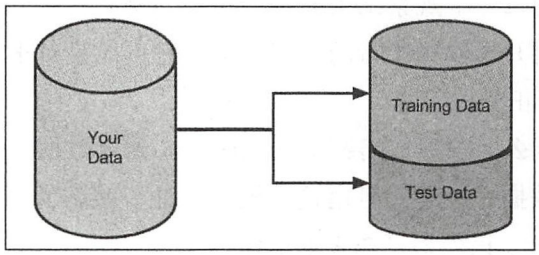
为了防止过度拟合，需要验证我们的模型。有多种方法可以执行验证，这里将介绍其中的几个。

每当你制作一个模型时，都需要确保已经验证了你的模型，并且理解它将如何推广到新数据。

3.2.1 训练和测试集

防止过度拟合的第一种方法是在一部分数据集上训练或拟合模型，然后在数据集的另一部分测试或评估该模型。训练模型通常包括参数化组成模型的一个或多个函数，例如期望值预测函数。然后，可以使用之前讨论的一个或多个评估指标来评估此训练模型。比较重要的是，不要用训练模型的同一份数据去测试 / 评估模型。

通过预留部分数据用于测试，可以模拟模型看到新数据的场景。也就是说，该模型是基于未用于参数化模型的数据进行预测。



许多人开始会将数据分为 80% 的训练集和 20% 的测试集（80/20 分组）。但是，会看到不同的人以不同的比例分割他们的数据集。测试与训练数据的比例取决于所拥有的数据类型和数量以及正在尝试训练的模型。一般来说，需要确保训练数据和测试数据能准确代表更大规模的数据。

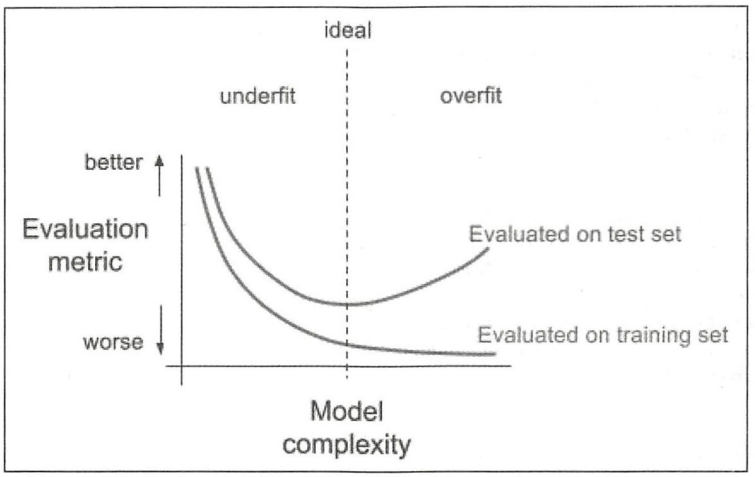
例如，如果试图预测 A、B 和 C 几种不同类别中的一种，则不希望训练数据仅

仅供非商业用途或交流学习使用

包含与 A 和 B 相对应的观察值。对这样的数据集进行训练的模型可能只能预测 A 和 B 类别。同样，不希望测试集中包含某些类别的子集或人为加权的类别比例。这很容易发生，取决于数据是如何生成的。

另外，需要确保有足够多的训练数据用来减少决定参数的变异性，因为这个参数会反复参与计算。如果训练数据点太少或训练数据点采样不准确，则模型训练可能会产生具有很大变异性的参数，或者可能无法数值收敛。以上这些迹象表明你的模型缺乏预测能力。

通常情况下，随着模型复杂性的增加，将有助于提高训练数据的评估指标，但同时，会降低测试数据的评估指标。当测试数据的评估指标开始变差时，模型就会开始过度拟合。理想情况是，提高模型复杂度直到测试评估指标开始下降的拐点。另一种方式（非常适合本书中建立模型的一般理念）是需要最可解释的模型（或简单模型），它可以产生有价值的结果。



快速将数据集分成训练和测试集的一种方法是使用 `github.com/kniren/gota/data-frame`。让我们使用数据集来展示这一点，其中包括一些关于医疗患者的匿名信息以及相应的疾病和糖尿病进展指标：

```
age,sex,bmi,map,tc,ldl,hdl,tch,ltg,glu,y
0.0380759064334,0.0506801187398,0.0616962065187,0.021872354995,-0.044223498
4244,-0.0348207628377,-0.043400845652,-0.00259226199818,0.0199084208763,-0.
0176461251598,151.0
-0.00188201652779,-0.044641636507,-0.0514740612388,-0.0263278347174,-0.0084
4872411122,-0.0191633397482,0.0744115640788,-0.0394933828741,-0.06832974362
44,-0.0922040496268,75.0
0.0852989062967,0.0506801187398,0.0444512133366,-0.00567061055493,-0.045599
4512826,-0.0341944659141,-0.0323559322398,-0.00259226199818,0.0028637705189
4,-0.0259303389895,141.0
-0.0890629393523,-0.044641636507,-0.0115950145052,-0.0366564467986,0.012190
5687618,0.0249905933641,-0.0360375700439,0.0343088588777,0.0226920225667,-0
.00936191133014,206.0
```

仅供非商业用途或交流学习使用

```
0.00538306037425,-0.044641636507,-0.0363846922045,0.021872354995,0.00393485
161259,0.0155961395104,0.00814208360519,-0.00259226199818,-0.0319914449414,
-0.0466408735636,135.0
...
```

可以在这里检索这个数据集：https://archive.ics.uci.edu/ml/datasets/diabetes。

为了使用 github.com/kniren/gota/dataframe 分割这个数据，可以执行以下操作（将分割的训练和测试数据保存到各自的 CSV 文件中）：

```
// Open the diabetes dataset file.
f, err := os.Open("diabetes.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a dataframe from the CSV file.
// The types of the columns will be inferred.
diabetesDF := dataframe.ReadCSV(f)

// Calculate the number of elements in each set.
trainingNum := (4 * diabetesDF.Nrow()) / 5
testNum := diabetesDF.Nrow() / 5
if trainingNum+testNum < diabetesDF.Nrow() {
    trainingNum++
}

// Create the subset indices.
trainingIdx := make([]int, trainingNum)
testIdx := make([]int, testNum)

// Enumerate the training indices.
for i := 0; i < trainingNum; i++ {
    trainingIdx[i] = i
}

// Enumerate the test indices.
for i := 0; i < testNum; i++ {
    testIdx[i] = trainingNum + i
}

// Create the subset dataframes.
trainingDF := diabetesDF.Subset(trainingIdx)
testDF := diabetesDF.Subset(testIdx)

// Create a map that will be used in writing the data
// to files.
setMap := map[int]dataframe.DataFrame{
    0: trainingDF,
    1: testDF,
}

// Create the respective files.
for idx, setName := range []string{"training.csv", "test.csv"} {
    // Save the filtered dataset file.
    f, err := os.Create(setName)
```

仅供非商业用途或交流学习使用

```
    if err != nil {
        log.Fatal(err)
    }

    // Create a buffered writer.
    w := bufio.NewWriter(f)

    // Write the dataframe out as a CSV.
    if err := setMap[idx].WriteCSV(w); err != nil {
        log.Fatal(err)
    }
}
```

运行这个会得到如下结果：

```
$ go build
$ ./myprogram
$ wc -l *.csv
   443 diabetes.csv
    89 test.csv
   355 training.csv
   887 total
```

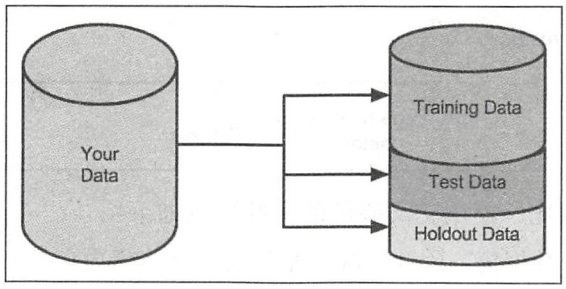
3.2.2　保留集

我们正在取得进展，以确保模型使用训练和测试集进行一般化。但是，想象一下如下的情况：

1. 我们根据训练集开发了模型的第一个版本。
2. 我们在测试集上测试了模型的第一个版本。
3. 我们对测试集的结果不满意，因此循环回到步骤 1 并重复。

这个过程可能看起来合乎逻辑，但你可能已经看到由此过程导致的问题。通过反复将模型暴露给测试集，实际上是过度拟合模型到测试数据。

有几种方法可以解决这种额外的过度拟合问题。首先是通过简单地创建另一个分割数据集，称为**保留集**（也称为**验证集**）。所以，现在会有一套训练集，测试集和保留集。这有时被称为三数据集验证，原因很明显。



请记住，要真正了解模型的一般性能，保留集绝不能用于训练和测试。应该保

仅供非商业用途或交流学习使用

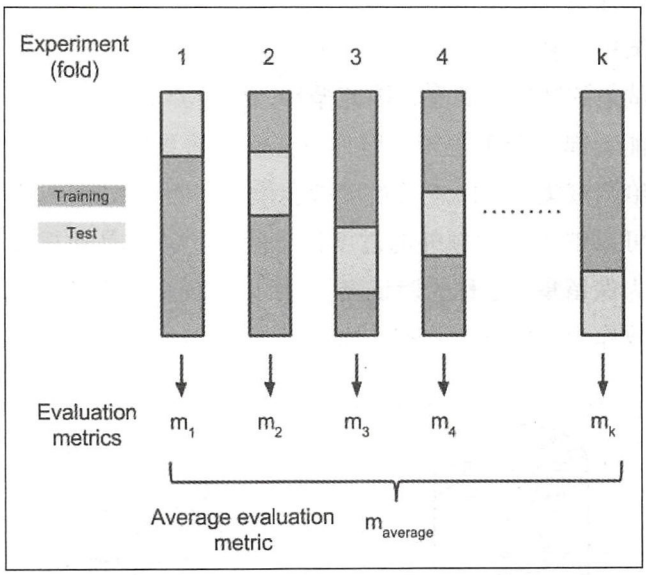
留这个数据集，用于在完成模型训练、模型调整和在测试数据集上获得可接受的性能之后的验证。

你可能想知道如何管理随时间推移的分割数据集，并恢复用于训练或测试某些模型的不同数据集。当尝试在机器学习工作流程中保持完整性时，数据的“出处”至关重要。这也正是 Pachyderm 的数据版本化（在第 1 章中介绍）所解决的问题。我们将在后面第 9 章中看到这一点。

3.2.3 交叉验证

除了保留一个保留集用来验证以外，交叉验证是验证模型一般性的常用技术。在交叉验证或 k-fold 交叉验证中，实际上会将数据集随机分为 k 份，然后应用到不同的训练和测试组合中。可以把这些看作 k 个实验。

一旦完成了每次拆分，就可以根据该拆分的训练数据对模型进行训练，然后根据该拆分的测试数据对其进行评估。此过程会为数据的每个随机拆分生成评估度量结果。然后，可以对这些评估指标进行平均，以获得一个总体评估指标，该评估指标比模型指标本身的任何一个评估指标更为通用。还可以查看评估指标中的差异来了解各种实验的稳定性。下图显示了此过程：



与数据集验证相比，使用交叉验证的一些优点如下：

- 正在使用整个数据集，因此实际上是将模型暴露给更多的训练样本和测试样本
- 已经为交叉验证编写了很多便利的函数和软件包
- 有助于防止选择单个验证集可能导致的偏差

仅供非商业用途或交流学习使用

github.com/sjwhitworth/golearn 是一个 Go 软件包，它为交叉验证提供了一些便利功能。实际上，github.com/sjwhitworth/golearn 包含了一些机器学习函数，稍后会在本书中介绍，但现在，来看看可用于交叉验证的函数。

如果查看 github.com/sjwhitworth/golearn/evaluation 软件包的 Godocs，将看到以下用于交叉验证的函数：

```
func GenerateCrossFoldValidationConfusionMatrices(data base.FixedDataGrid,
cls base.Classifier, folds int) ([]ConfusionMatrix, error)
```

这个函数实际上可以与各种模型一起使用，但这里是一个使用决策树模型的例子（暂时不用担心模型的细节）：

```
// Define the decision tree model.
tree := trees.NewID3DecisionTree(param)

// Perform the cross validation.
cfs, err := evaluation.GenerateCrossFoldValidationConfusionMatrices(myData,
tree, 5)
if err != nil {
    panic(err)
}

// Calculate the metrics.
mean, variance := evaluation.GetCrossValidatedMetric(cfs,
evaluation.GetAccuracy)
stdev := math.Sqrt(variance)

// Output the results to standard out.
fmt.Printf("%0.2f\t\t%.2f (+/- %.2f)\n", param, mean, stdev*2)
```

3.3　参考书目

评估：

- 有关过度拟合的文章：http://scott.fortmann-roe.com/docs/MeasuringError.html
- 关于偏差 – 方差权衡的文章：http://scott.fortmann-roe.com/docs/BiasVariance.html
- 分类评估指标比较：https://en.wikipedia.org/wiki/Precision_and_recall
- gonum.org/v1/gonum/stat 文档：https://godoc.org/gonum.org/v1/gonum/stat
- github.com/sjwhitworth/golearn/evaluation 文档：https://godoc.org/github.com/sjwhitworth/golearn/evaluation

验证：

- github.com/kniren/gota/dataframe 文档：https://godoc.org/github.com/kniren/gota/dataframe

仅供非商业用途或交流学习使用

- `github.com/sjwhitworth/golearn/evaluation` 文档：https://godoc.org/github.com/sjwhitworth/golearn/evaluation

3.4 小结

选择合适的评估指标并制定评估 / 验证程序是任何机器学习项目的重要组成部分。你已经了解了许多相关的评估指标，以及如何使用保留集和 / 或交叉验证以避免过度拟合。在下一章中，将开始研究机器学习模型，并使用线性回归构建第一个模型！

仅供非商业用途或交流学习使用

第 4 章
回　　归

我们将探索的第一组机器学习技术被称为**回归**（regression）。回归是一个过程，通过回归可以了解到一个变量（例如销售额）如何随着另一个变量（例如用户数）发生变化。这些技术在它们自身领域内是有用的。然而，回归也是讨论机器学习的一个很好切入点，因为回归是其他更复杂技术的基础，我们稍后会讨论这些技术。

一般来说，机器学习的回归技术与连续值（例如，股票价格、温度或者疾病进展）预测有关。在下一章会涉及的**分类**（classification）就与离散变量，或者某个类别的离散集（例如，欺诈 / 非欺诈，坐下 / 站立 / 跑动，热狗 / 非热狗）有关。如前所述，回归技术作为分类算法的一部分，会贯穿整个机器学习，但是本章重点是连续数值预测的基本应用。

4.1　理解回归模型的术语

如前文所述，回归本身是分析一个变量同另一个变量之间关系的过程，但是在机器学习中仍然使用了一些术语来描述这些变量，以及与回归有关的各种类型和过程。

- **响应或者因变量**：这些术语等价于基于一个或多个变量进行预测的另一个变量。这个变量通常标记为 y。
- **解释变量**、**自变量**、**特征**、**属性**或者**回归量**：这些术语等价于用来预测响应的变量。这些变量经常标记为为 x 或者 x_1、x_2，等等。
- **线性回归**：这种类型的回归假定因变量线性依赖自变量（也就是说，按照线性方程）。
- **非线性回归**：这种类型的回归假定因变量非线性依赖自变量（例如多项式或

仅供非商业用途或交流学习使用

者指数）。

- **多元回归**：具有多个自变量的回归。
- **拟合和训练**：参数化一个模型（例如回归模型）的过程，用以预测特定因变量。
- **预测**：使用参数化模型（例如回归模型）预测一个特定因变量的过程。

上述这些术语会在本书其他部分涉及，并被用在回归和其他语境中。

4.2 线性回归

线性回归是最简单的机器学习模型之一。但是不应以任何理由忽略该模型。正如前面所提到的，线性回归模型是其他模型的基础，并且有一些非常重要的优点。

正如贯穿本书的观点，完整性在机器学习的应用中至关重要，模型越简单、越容易解释，就更容易维持完整性。除此之外，因为模型简单和易解释，可以更容易理解变量之间的推导关系，并且可以从心理上检查开发工作。正如 Mike Lee Williams 在 Fast Forward Labs 所说的（http://blog.fastforwardlabs.com/2017/08/02/interpretability.html）：

> 未来是算法的，可解释的模型提供了人类和智能机器之间安全、高产、最终更加协作的关系。

线性回归模型是可解释的，因此可以提供给数据科学家安全和有效的选择。当寻找一个用于预测连续变量的模型时，如果在数据和问题允许的情况下，不妨考虑和尝试一下线性回归（甚至多元回归）。

4.2.1 线性回归概述

在线性回归中，可以尝试使用线性方程，用自变量 x 为因变量 y 建模。

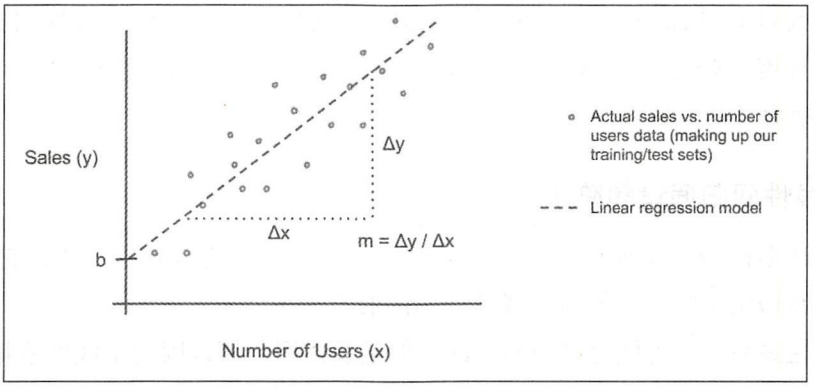
$$y=mx+b$$

在这里，m 是线段的斜率，b 是截距。假如想根据网站每天的用户数量（number of users）对每日销售额（sales）进行建模。如果用线性回归来做，需要确定 m 和 b，然后用如下方程式预测销售额。

$$sales=m*(number\ of\ users)+b$$

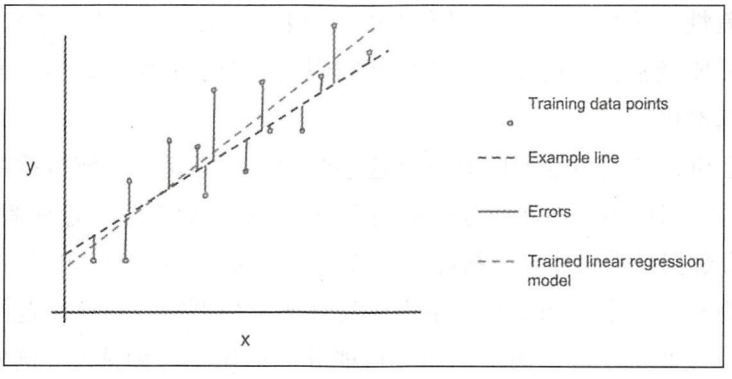
因此，我们的模型训练就是这个参数化的函数。输入**用户数量**，然后得到预测的**销售额**，如下所示。

仅供非商业用途或交流学习使用



线性回归模型的训练或拟合包含确定 m 和 b 的值，所以该方程式具有预测响应的能力。虽然有很多方法能够用来确定 m 和 b，但是最常见和简单的方法是**普通最小二乘法**（OLS）。

为了使用 OLS 方式找到 m 和 b，首先为 m 和 b 赋值用来创建第一个示例直线，然后测量每个已知点（例如，从测试集中）到示例直线垂直方向的距离。这个距离称为**误差**（error）或者**残差**（residual），类似于第 3 章并用以下图表进行了阐述。



下一步，将这些误差的平方和相加。

$$\frac{error_1^2+error_2^2+\cdots+error_N^2}{N}$$

调整 m 和 b 直到误差的平方和数值最小。换句话说，线性回归直线是误差平方和最小的直线。

有许多方法可以找到误差平方和最小的直线，对于 OLS，这个直线可通过分析找到。然而，有一个非常流行和常见的优化技术可用于计算最小化误差平方和，它就是**梯度下降法**。这个方法在应用方面更加高效，计算方面（例如在内存方面）更有优势，并且比分析解决方案更加灵活。

梯度下降法在附录中有更详细的解释，因此在这里不做冗长的讨论。可以这么

仅供非商业用途或交流学习使用

说，许多线性和其他回归的实现都将梯度下降法用于拟合和训练线性回归直线上。实际上，梯度下降法在机器学习中非常普遍，并且也为复杂的建模技术（如深度学习）提供动力。

4.2.2 线性回归假设和陷阱

正如许多机器学习模型那样，线性回归并不能涵盖所有场景，并且它对数据和数据的关系做出了特定的假设。线性回归的假设如下所示：

- **线性关系**：这可能是显而易见的，但是线性回归假设因变量线性依赖（通过线性方程）自变量。如果这种关系是非线性的，线性回归的性能可能表现不佳。
- **正态性**：这个假设意味着变量是正态分布的（看起来像钟形）。本章稍后会再次讨论该属性，并讨论当遇到非正态分布变量时的一些权衡和选项。
- **非多重共线性**：多重共线性是一个奇特的术语，它意味着自变量并不是真正独立，它们以某种方式相互依赖。
- **无自相关性**：自相关是另一个奇特的术语，它意味着变量依赖于自身或者自身的一些历史版本（例如一些可预测的时间序列）。
- **同方差性**：这可能是一系列术语中最奇特的词语，但是它意味着相对简单、并不需要经常担心的东西。线性回归假设自变量的所有取值在回归线附近的方差基本相同。

从技术上讲，使用线性回归时所有这些假设应该满足。了解数据的分布和行为是非常重要的。当使用线性回归分析示例中的数据时，将会研究这些假设。

作为一个数据科学家或者分析师，当应用线性回归时应牢记如下陷阱：

- 你在为某个特定范围的自变量训练线性回归模型时。对这个范围以外的数值预测应该非常小心，因为回归线可能并不适用于这种情况（例如，因变量在极值处可能开始表现为非线性）。
- 可能因为找到两个原本没有关系的变量之间的虚假关系，从而错误地采用线性回归模型。应当仔细检查以确保变量功能上相关是有内在逻辑原因的。
- 在某些拟合中，数据中的异常或极值可能偏离回归线，例如 OLS。有多种方法可以拟合对异常免疫的回归线，或者对异常敏感的回归线，例如最小正交二乘或者岭回归。

4.2.3 线性回归示例

为了说明线性回归，将举一个示例并创建第一个机器学习模型。这里将采用广告样本数据作为示例数据。它是如下所示的 .cvs 格式：

仅供非商业用途或交流学习使用

```
$ head Advertising.csv
TV,Radio,Newspaper,Sales
230.1,37.8,69.2,22.1
44.5,39.3,45.1,10.4
17.2,45.9,69.3,9.3
151.5,41.3,58.5,18.5
180.8,10.8,58.4,12.9
8.7,48.9,75,7.2
57.5,32.8,23.5,11.8
120.2,19.6,11.6,13.2
8.6,2.1,1,4.8
```

数据集包括一组代表广告渠道（电视（TV）、广播（Radio）和报纸（Newspaper））的支出以及相应销售额（Sales）。本例的目标是通过广告花费（自变量）的一个属性来构建销售额（因变量）模型。

4.2.3.1 分析数据

为了确保创建一个易于理解的模型或者至少是过程，并且确保能够直观地检查结果，需要在每个机器学习模型的构建之前进行数据分析。需要了解每个变量的分布，以及它们的范围和变化趋势。

要做到这一点，需要计算之前在第2章中提到的汇总统计数据。在这里，将利用 github.com/kniren/gota/dataframe 软件包中的内置方法，用一个操作为数据集中所有列计算汇总统计数据：

```
// Open the CSV file.
advertFile, err := os.Open("Advertising.csv")
if err != nil {
    log.Fatal(err)
}
defer advertFile.Close()

// Create a dataframe from the CSV file.
advertDF := dataframe.ReadCSV(advertFile)

// Use the Describe method to calculate summary statistics
// for all of the columns in one shot.
advertSummary := advertDF.Describe()

// Output the summary statistics to stdout.
fmt.Println(advertSummary)
```

编译和运行会得到如下结果：

```
$ go build
$ ./myprogram
[7x5] DataFrame

    column   TV          Radio     Newspaper  Sales
 0: mean     147.042500 23.264000 30.554000  14.022500
 1: stddev   85.854236  14.846809 21.778621  5.217457
 2: min      0.700000   0.000000  0.300000   1.600000
 3: 25%      73.400000  9.900000  12.600000  10.300000
```

仅供非商业用途或交流学习使用


```
4: 50%      149.700000 22.500000 25.600000  12.900000
5: 75%      218.500000 36.500000 45.100000  17.400000
6: max      296.400000 49.600000 114.000000 27.000000
   <string> <float>    <float>   <float>    <float>
```

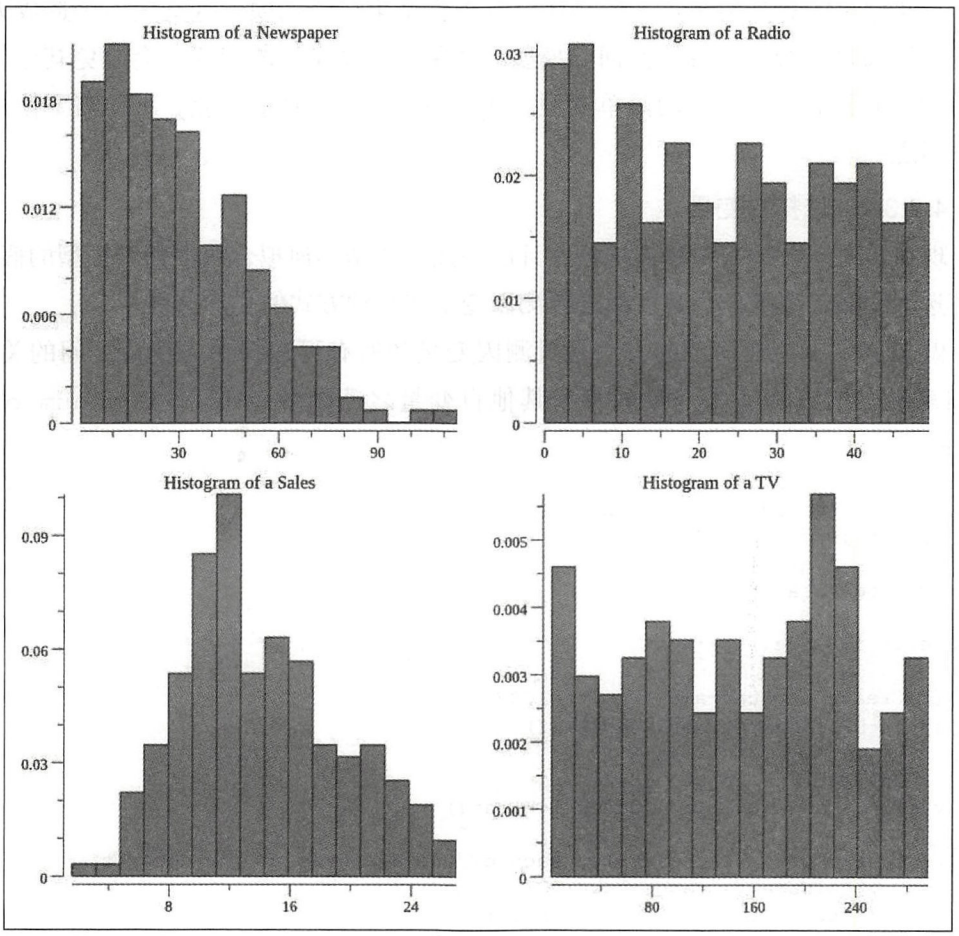
如你所见，这会以良好的表格形式打印出所有汇总统计数据，其中包括均值、标准差、极小值、极大值、25%/75% 分位数，以及中位数（或者 50% 分位数）。

这些数值给了一个良好的数字参考，这在训练回归模型中也会看到。但是这并没有提供数据的一个很好的直观效果。为此，将为每列中的数值创建一个直方图：

```
// Open the advertising dataset file.
f, err := os.Open("Advertising.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a dataframe from the CSV file.
advertDF := dataframe.ReadCSV(f)

// Create a histogram for each of the columns in the dataset.
for _, colName := range advertDF.Names() {

    // Create a plotter.Values value and fill it with the
    // values from the respective column of the dataframe.
    plotVals := make(plotter.Values, advertDF.Nrow())
    for i, floatVal := range advertDF.Col(colName).Float() {
        plotVals[i] = floatVal
    }

    // Make a plot and set its title.
    p, err := plot.New()
    if err != nil {
        log.Fatal(err)
    }
    p.Title.Text = fmt.Sprintf("Histogram of a %s", colName)

    // Create a histogram of our values drawn
    // from the standard normal.
    h, err := plotter.NewHist(plotVals, 16)
    if err != nil {
        log.Fatal(err)
    }

    // Normalize the histogram.
    h.Normalize(1)

    // Add the histogram to the plot.
    p.Add(h)

    // Save the plot to a PNG file.
    if err := p.Save(4*vg.Inch, 4*vg.Inch, colName+"_hist.png"); err != nil
{
        log.Fatal(err)
    }
}
```

仅供非商业用途或交流学习使用

这个程序将为每个直方图创建一个 .png 图像：



现在，查看直方图和计算的汇总统计数据。需要考虑是否符合线性回归的假设。特别是，看到并非所有变量都是正态分布（也就是说，钟形分布）。销售额可能有点钟形分布的样子，但是其他的看起来并非正态。

可以使用例如 quantile-quantile（q-q）图的统计学工具来确定分布同正态分布的接近程度。甚至可以采用统计测试用于确定变量服从正态分布的概率。但是大多数时候，可以从直方图获得一个总体思路。

现在必须做出决定。至少有一些数据从技术上讲并不符合线性回归模型的假设。可以执行以下任意操作：

- 试着转换变量（例如幂转换）用以符合正态分布，然后应用转换后的变量到线性回归模型中。这个选择的好处是能够在模型的假设下运作。缺点是让模型变得更难理解，而且不太容易解释。
- 获取不同的数据用以解决这个问题。

仅供非商业用途或交流学习使用

- 用线性回归的假设忽略问题，然后尝试创建模型。

对此你可能有其他观点，但是建议首先尝试第三个选项。这选项不会造成太大损害，因为可以很快训练线性回归模型。如果最后模型性能很好，会避免其他复杂的问题，并拥有一个不错的简单模型。如果最后模型的性能很差，可能需要使用其他选项之一。

4.2.3.2 选择自变量

现在，对于数据有一些直觉，并且已经接近数据如何拟合线性回归模型的假设。如何选择用某个变量作为自变量去预测因变量和每种方式的平均分数？

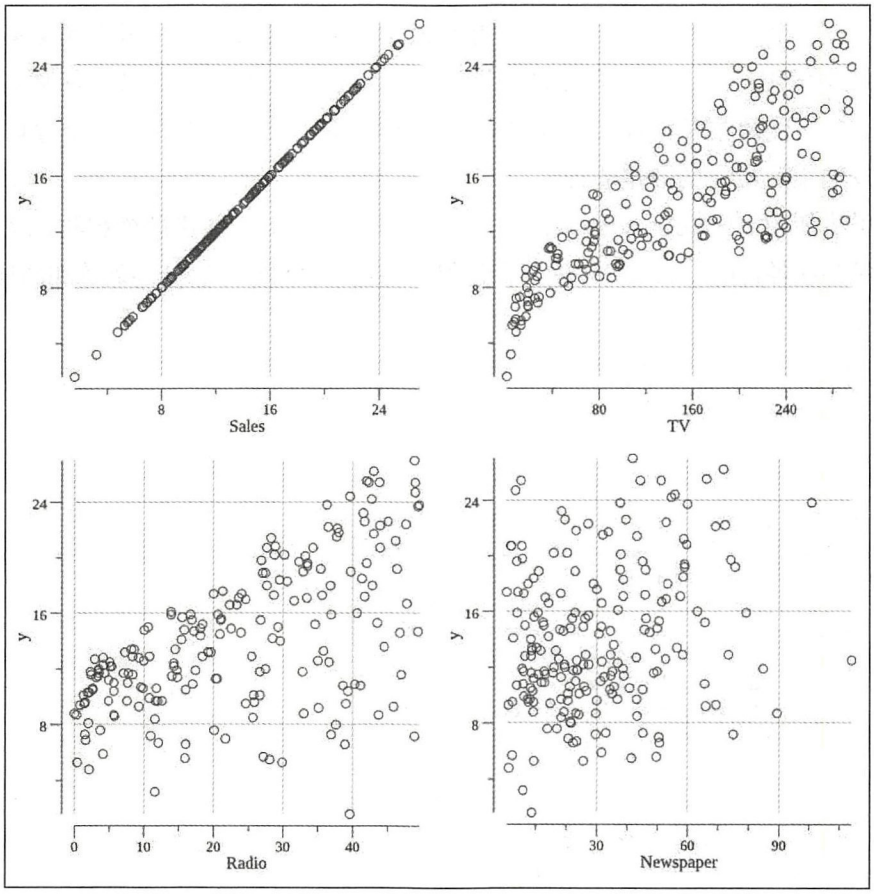
做出这一决定最简单的方法是目测因变量和所有可选择的自变量之间的关系。详细来说，可以构造因变量同每个其他自变量之间的散点图（使用 gonum.org/v1/plot）：

```
// Open the advertising dataset file.
f, err := os.Open("Advertising.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a dataframe from the CSV file.
advertDF := dataframe.ReadCSV(f)

// Extract the target column.
yVals := advertDF.Col("Sales").Float()

// Create a scatter plot for each of the features in the dataset.
for _, colName := range advertDF.Names() {
    // pts will hold the values for plotting
    pts := make(plotter.XYs, advertDF.Nrow())

    // Fill pts with data.
    for i, floatVal := range advertDF.Col(colName).Float() {
        pts[i].X = floatVal
        pts[i].Y = yVals[i]
    }

    // Create the plot.
    p, err := plot.New()
    if err != nil {
        log.Fatal(err)
    }
    p.X.Label.Text = colName
    p.Y.Label.Text = "y"
    p.Add(plotter.NewGrid())

    s, err := plotter.NewScatter(pts)
    if err != nil {
        log.Fatal(err)
    }
    s.GlyphStyle.Radius = vg.Points(3)
```

仅供非商业用途或交流学习使用


```
    // Save the plot to a PNG file.
    p.Add(s)
    if err := p.Save(4*vg.Inch, 4*vg.Inch, colName+"_scatter.png"); err !=
nil {
        log.Fatal(err)
    }
}
```

这将创建如下的散点图：



当查看这些散点图时，需要考虑如何推断哪些属性（TV，Radio 和 / 或 Newspaper）同因变量 Sales 线性相关。也就是说，能否在任何一个对应属性的散点图上画一条符合销售额趋势的直线？这种方式不是万能的。并且对于给定问题使用全部属性是不现实的。

在这种情况下，Radio 和 TV 看起来同 Sales 有一定线性相关性。Newspaper 可能同 Sales 有一定相关，但是相关性并不明显。同 TV 的线性关系似乎最明显，因此将 TV 选做线性回归模型的自变量。这会得到如下线性回归公式：

$$Sales=m\ TV+b$$

另一件值得注意的事情是，TV 变量并不是严格的同方差，同方差在之前作为线性回归的一个假设做过讨论。这需要引起注意（并且可能需要在项目中进行记录），但是在这里将继续研究能否创建有某些预测能力的线性回归模型。作为一种可能的解释，如果模型性能不佳，随时可以重新审视这个假设。

4.2.3.3 创建训练和测试集合

为了避免过度拟合并确保模型具有通用性，这里将把数据集合拆分成训练集和测试集，正如第 3 章中提到的。不必为设置一个保留集而困扰，因为保留集只会在模型训练中使用一次，无须在测试和训练中来回迭代。然而，如果正在调试各种因变量和 / 或迭代调整任何模型的参数，是需要创建一个保留集的，这个保留集用于模型开发结束后最终的数据验证。

这里将使用 `github.com/kniren/gota/dataframe` 创建训练和测试集合，然后将其保存在各自的 `.csv` 文件中。在这里，将使用 80/20 方法分割训练和测试数据：

```
// Open the advertising dataset file.
f, err := os.Open("Advertising.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a dataframe from the CSV file.
// The types of the columns will be inferred.
advertDF := dataframe.ReadCSV(f)

// Calculate the number of elements in each set.
trainingNum := (4 * advertDF.Nrow()) / 5
testNum := advertDF.Nrow() / 5
if trainingNum+testNum < advertDF.Nrow() {
    trainingNum++
}

// Create the subset indices.
trainingIdx := make([]int, trainingNum)
testIdx := make([]int, testNum)

// Enumerate the training indices.
for i := 0; i < trainingNum; i++ {
    trainingIdx[i] = i
}

// Enumerate the test indices.
for i := 0; i < testNum; i++ {
    testIdx[i] = trainingNum + i
}

// Create the subset dataframes.
trainingDF := advertDF.Subset(trainingIdx)
testDF := advertDF.Subset(testIdx)
```

仅供非商业用途或交流学习使用

```
// Create a map that will be used in writing the data
// to files.
setMap := map[int]dataframe.DataFrame{
    0: trainingDF,
    1: testDF,
}

// Create the respective files.
for idx, setName := range []string{"training.csv", "test.csv"} {

    // Save the filtered dataset file.
    f, err := os.Create(setName)
    if err != nil {
        log.Fatal(err)
    }

    // Create a buffered writer.
    w := bufio.NewWriter(f)

    // Write the dataframe out as a CSV.
    if err := setMap[idx].WriteCSV(w); err != nil {
        log.Fatal(err)
    }
}
```

该代码将输出稍后会用到的训练和测试集合：

```
$ wc -l *.csv
  201 Advertising.csv
   41 test.csv
  161 training.csv
  403 total
```

这里使用的数据并没有以任何形式分组或排序。但是，如果你处理数据时按照响应、日期或者任何方式来排序，将数据随机分割为训练和测试集就非常重要，否则，测试和训练集可能仅包括一定范围的响应，可能会受到时间 / 日期等人为因素影响。

4.2.3.4　训练模型

接下来，将实际训练或拟合线性回归模型。如果你还记得，这意味着将寻找线段的斜率（m）和截距（b），使得误差的平方和最小。为了进行这个训练，将采用来自 Sajari 的一个非常实用的软件包：`github.com/sajari/regression`。Sajari 是一家非常依赖 Go 和机器学习的网络搜索公司。它们在实际生产环境中使用 `github.com/sajari/regression`。

为了使用 `github.com/sajari/regression` 训练一个回归模型，需要初始化 `regression.Regression` 设置一些标记，然后用标记的训练数据点填充 `regression.Regression` 数值。之后，训练线性回归模型就像在 `regression.Regression` 上调用 `Run()` 方法一样简单：

仅供非商业用途或交流学习使用

```
// Open the training dataset file.
f, err := os.Open("training.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a new CSV reader reading from the opened file.
reader := csv.NewReader(f)

// Read in all of the CSV records
reader.FieldsPerRecord = 4
trainingData, err := reader.ReadAll()
if err != nil {
    log.Fatal(err)
}

// In this case we are going to try and model our Sales (y)
// by the TV feature plus an intercept. As such, let's create
// the struct needed to train a model using github.com/sajari/regression.
var r regression.Regression
r.SetObserved("Sales")
r.SetVar(0, "TV")

// Loop of records in the CSV, adding the training data to the regression
value.
for i, record := range trainingData {

    // Skip the header.
    if i == 0 {
        continue
    }

    // Parse the Sales regression measure, or "y".
    yVal, err := strconv.ParseFloat(record[3], 64)
    if err != nil {
        log.Fatal(err)
    }

    // Parse the TV value.
    tvVal, err := strconv.ParseFloat(record[0], 64)
    if err != nil {
        log.Fatal(err)
    }

    // Add these points to the regression value.
    r.Train(regression.DataPoint(yVal, []float64{tvVal}))
}

// Train/fit the regression model.
r.Run()

// Output the trained model parameters.
fmt.Printf("\nRegression Formula:\n%v\n\n", r.Formula)
```

编译并运行上述代码可以将已训练的线性回归公式打印到 stdout：

仅供非商业用途或交流学习使用

```
$ go build
$ ./myprogram

Regression Formula:
Predicted = 7.07 + TV*0.05
```

在这里，可以看到，该软件包确定了线性回归线，截距为 7.07，斜率为 0.05。可以进行一些感觉上的检查，因为在之前的散点图中可以看到 TV 与 Sales 的对应关系是右上趋势（即正相关）。这意味着斜率在公式中应该是正值，这是正确的。

4.2.3.5　评估训练的模型

现在需要衡量模型的表现，看看是否真的有能力使用 TV 作为自变量来预 Sales。为此，可以加载测试集，使用训练好的模型对每个测试示例进行预测，然后计算第 3 章中讨论的评估指标之一。

对于这个问题，使用平均绝对误差（MAE）作为评估指标。这似乎是合理的，因为它可以直接与 Sales 数值进行比较，不必太担心离群值或极端值。

要使用训练的 `regression.Regression` 值计算预测的 Sales 数值，只需要解析测试集中的值并在 `regression.Regression` 值上调用 `Predict()` 方法。然后，将获得这些预测值与观测值的差值，得到差值的绝对值，然后将所有绝对值相加得到 MAE：

```
// Open the test dataset file.
f, err = os.Open("test.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a CSV reader reading from the opened file.
reader = csv.NewReader(f)

// Read in all of the CSV records
reader.FieldsPerRecord = 4
testData, err := reader.ReadAll()
if err != nil {
    log.Fatal(err)
}

// Loop over the test data predicting y and evaluating the prediction
// with the mean absolute error.
var mAE float64
for i, record := range testData {

    // Skip the header.
    if i == 0 {
        continue
    }
```

仅供非商业用途或交流学习使用

```
        // Parse the observed Sales, or "y".
        yObserved, err := strconv.ParseFloat(record[3], 64)
        if err != nil {
            log.Fatal(err)
        }

        // Parse the TV value.
        tvVal, err := strconv.ParseFloat(record[0], 64)
        if err != nil {
            log.Fatal(err)
        }

        // Predict y with our trained model.
        yPredicted, err := r.Predict([]float64{tvVal})

        // Add the to the mean absolute error.
        mAE += math.Abs(yObserved-yPredicted) / float64(len(testData))
    }

    // Output the MAE to standard out.
    fmt.Printf("MAE = %0.2f\n\n", mAE)
```

编译并运行此评估会得出以下结果：

```
$ go build
$ ./myprogram

Regression Formula:
Predicted = 7.07 + TV*0.05

MAE = 3.01
```

我们如何知道 MAE=3.01 是好还是不好？这也是为什么拥有良好的数据心理模型很重要。如果你记得，这里已经计算出销售的平均值、范围和标准差。平均销售价值为 14.02，标准差为 5.21。因此，MAE 低于销售额的标准差，约为平均值的 20%，此模型具有一定的预测能力。

所以，恭喜！现在已经建立了第一个具有预测能力的机器学习模型！

为了更好地理解这个模型的表现，还可以创建一个图表来帮助可视化线性回归线。这可以通过 gonum.org/v1/plot 完成。首先，创建一个预测函数，使能够在不导入 github.com/sajari/regression 的情况下进行预测。这里提供了一个轻量级的内存版训练模型：

```
// predict uses our trained regression model to made a prediction.
func predict(tv float64) float64 {
    return 7.07 + tv*0.05
}
```

然后，可以创建回归线的可视化：

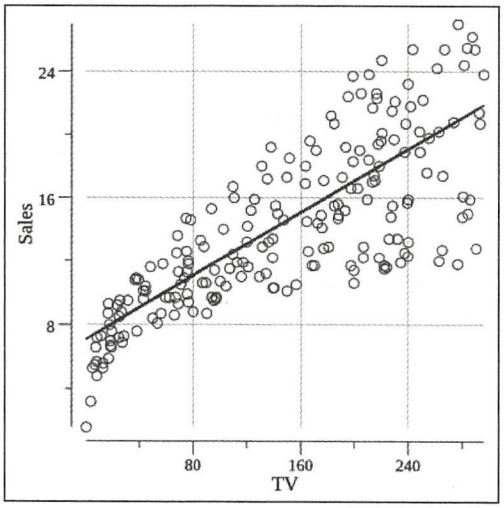
```
// Open the advertising dataset file.
f, err := os.Open("Advertising.csv")
if err != nil {
```

仅供非商业用途或交流学习使用

```
    log.Fatal(err)
}
defer f.Close()

// Create a dataframe from the CSV file.
advertDF := dataframe.ReadCSV(f)

// Extract the target column.
yVals := advertDF.Col("Sales").Float()

// pts will hold the values for plotting.
pts := make(plotter.XYs, advertDF.Nrow())

// ptsPred will hold the predicted values for plotting.
ptsPred := make(plotter.XYs, advertDF.Nrow())

// Fill pts with data.
for i, floatVal := range advertDF.Col("TV").Float() {
    pts[i].X = floatVal
    pts[i].Y = yVals[i]
    ptsPred[i].X = floatVal
    ptsPred[i].Y = predict(floatVal)
}

// Create the plot.
p, err := plot.New()
if err != nil {
    log.Fatal(err)
}
p.X.Label.Text = "TV"
p.Y.Label.Text = "Sales"
p.Add(plotter.NewGrid())

// Add the scatter plot points for the observations.
s, err := plotter.NewScatter(pts)
if err != nil {
    log.Fatal(err)
}
s.GlyphStyle.Radius = vg.Points(3)

// Add the line plot points for the predictions.
l, err := plotter.NewLine(ptsPred)
if err != nil {
    log.Fatal(err)
}
l.LineStyle.Width = vg.Points(1)
l.LineStyle.Dashes = []vg.Length{vg.Points(5), vg.Points(5)}

// Save the plot to a PNG file.
p.Add(s, l)
if err := p.Save(4*vg.Inch, 4*vg.Inch, "regression_line.png"); err != nil {
    log.Fatal(err)
}
```

编译并运行后会得到如下图表：

仅供非商业用途或交流学习使用



正如你所见，训练的线性回归线符合实际数据点的线性趋势。这是另一个视觉确认，我们正处于正确的轨道上！

4.3 多元线性回归

线性回归不仅仅是只有一个自变量的简单公式。多元线性回归与之前讨论过的类似，但在这里有多个自变量（x_1、x_2 等）。在这种情况下，使用如下简单的方程式：

$$y=m_1x_1+m_1x_2+\cdots+m_{N\times N}+b$$

这里，x 是各种自变量，m 是与这些自变量相关的各种斜率。这里仍然还有一个截距 b。

多元线性回归有点难以形象化和思考，因为这不再是可以在两个维度上可视化的线。它是二维，三维或更多维度的线性表面。但是，许多用于一元线性回归的技术同样也会贯穿始终。

多元线性回归与常规线性回归具有相同的假设。但是，应该记住一些容易遇到的问题：

- **过度拟合**（overfitting）：通过向模型添加越来越多的自变量增加了模型的复杂性，这有面临过度拟合的风险。处理这个问题的一种技术称为**正则化**（regularization），建议研究一下。正则化会在模型中创建一个惩罚项，换句话说就是创建一个衡量模型复杂性的函数，这样做有助于保证此模型可控。
- **相对比例**（relative scale）：在某些情况下，一个自变量的量级与另一个自变量的量级不同。其中较大的可以消除较小的影响，因此需要考虑变量正则化。

考虑到这一点，可以尝试将销售模型从线性回归模型扩展到多元回归模型。回

仅供非商业用途或交流学习使用

顾前一部分的散点图，可以看到 Radio 似乎也同 Sales 线性相关，所以尝试创建一个如下的多元线性回归模型：

$$Sales=m_1\ TV+m_2\ Radio+b$$

要使用 github.com/sajari/regression 执行此操作，只需在 regression.Regression 值中标注另一个变量，并确保这些值在训练数据点中配对。然后运行回归，看看公式结果如何：

```
// Open the training dataset file.
f, err := os.Open("training.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()
// Create a new CSV reader reading from the opened file.
reader := csv.NewReader(f)

// Read in all of the CSV records
reader.FieldsPerRecord = 4
trainingData, err := reader.ReadAll()
if err != nil {
    log.Fatal(err)
}

// In this case we are going to try and model our Sales
// by the TV and Radio features plus an intercept.
var r regression.Regression
r.SetObserved("Sales")
r.SetVar(0, "TV")
r.SetVar(1, "Radio")

// Loop over the CSV records adding the training data.
for i, record := range trainingData {

    // Skip the header.
    if i == 0 {
        continue
    }

    // Parse the Sales.
    yVal, err := strconv.ParseFloat(record[3], 64)
    if err != nil {
        log.Fatal(err)
    }

    // Parse the TV value.
    tvVal, err := strconv.ParseFloat(record[0], 64)
    if err != nil {
        log.Fatal(err)
    }

    // Parse the Radio value.
    radioVal, err := strconv.ParseFloat(record[1], 64)
    if err != nil {
        log.Fatal(err)
    }
```

仅供非商业用途或交流学习使用

```
        // Add these points to the regression value.
        r.Train(regression.DataPoint(yVal, []float64{tvVal, radioVal}))
    }

    // Train/fit the regression model.
    r.Run()

    // Output the trained model parameters.
    fmt.Printf("\nRegression Formula:\n%v\n\n", r.Formula)
```

编译和运行上述代码将得到如下回归公式：

```
$ go build
$ ./myprogram

Regression Formula:
Predicted = 2.93 + TV*0.05 + Radio*0.18
```

正如所看到的，回归公式现在包含额外项，代表自变量Radio。相比之前一元回归模型，截距数值也发生了改变。

可以像一元回归模型那样，使用Perdict方法测试这个模型：

```
// Open the test dataset file.
f, err = os.Open("test.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a CSV reader reading from the opened file.
reader = csv.NewReader(f)

// Read in all of the CSV records
reader.FieldsPerRecord = 4
testData, err := reader.ReadAll()
if err != nil {
    log.Fatal(err)
}

// Loop over the test data predicting y and evaluating the prediction
// with the mean absolute error.
var mAE float64
for i, record := range testData {

    // Skip the header.
    if i == 0 {
        continue
    }

    // Parse the Sales.
    yObserved, err := strconv.ParseFloat(record[3], 64)
    if err != nil {
        log.Fatal(err)
    }

    // Parse the TV value.
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
        tvVal, err := strconv.ParseFloat(record[0], 64)
        if err != nil {
            log.Fatal(err)
        }

        // Parse the Radio value.
        radioVal, err := strconv.ParseFloat(record[1], 64)
        if err != nil {
            log.Fatal(err)
        }

        // Predict y with our trained model.
        yPredicted, err := r.Predict([]float64{tvVal, radioVal})

        // Add the to the mean absolute error.
        mAE += math.Abs(yObserved-yPredicted) / float64(len(testData))
    }

    // Output the MAE to standard out.
    fmt.Printf("MAE = %0.2f\n\n", mAE)
```

运行此操作可为新多元回归模型计算得到以下 MAE：

```
$ go build
$ ./myprogram

Regression Formula:
Predicted = 2.93 + TV*0.05 + Radio*0.18

MAE = 1.26
```

新多元回归模型改进了原有的 MAE！ 现在基于广告花费预测 Sales 肯定性能良好，也可以在后续练习中尝试添加 Newspaper 到模型中，以了解模型性能如何受到影响。

请记住，当为模型添加更多的复杂性时，牺牲了简单性，并且可能有过度拟合的危险，因此只有模型性能提升能够创造更多价值的时候，才会去增加模型复杂性。

4.4 非线性和其他类型的回归

尽管在本章中专注于线性回归，但当然不会仅局限于使用线性公式进行回归。可以通过一个或多个非线性项（例如幂、指数或其他自变量转换）来为因变量建模。例如，可以通过一个多项式的许多 *TV* 项为 *Sales* 建模：

$$Sales=m_1\,TV+m_2\,TV^2+m_3\,TV^3+\cdots+b$$

但请记住，当增加这种复杂性时，又容易陷入过度拟合的危险之中。

在实施非线性回归方面，不能使用 github.com/sajari/regression，它仅

仅供非商业用途或交流学习使用

限于线性回归。但是，go-hep.org/x/hep/fit 允许拟合或训练某些非线性模型，而 Go 社区中还有大量人员已经、即将或正在开发其他非线性建模工具。

除了 OLS 之外，还有其他线性回归技术有助于克服与最小二乘线性回归相关的一些假设和弱点。这些包括**岭回归**（ridge regression）和**套索回归**（lasso regression）。这两种技术都对回归系数进行惩罚，以减轻自变量的多重共线性和非正态性的影响。

在 Go 实现方面，岭回归在 github.com/berkmancenter/ridge 中实现。与 github.com/sajari/regression 相反，自变量和因变量数据通过 gonum 矩阵输入到 github.com/berkmancenter/ridge。因此，为了说明这种方法，首先形成一个包含广告支出特征（TV、Radio 以及 Newspaper）的矩阵和一个包含 Sales 数据的矩阵。请注意，在 github.com/berkmancenter/ridge 中，如果希望在模型中有一个截距，需要在输入自变量矩阵中明确添加一列用于截距。此列中的每个值都是 1.0。

```
// Open the training dataset file.
f, err := os.Open("training.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()
// Create a new CSV reader reading from the opened file.
reader := csv.NewReader(f)
reader.FieldsPerRecord = 4

// Read in all of the CSV records
rawCSVData, err := reader.ReadAll()
if err != nil {
    log.Fatal(err)
}

// featureData will hold all the float values that will eventually be
// used to form our matrix of features.
featureData := make([]float64, 4*len(rawCSVData))
yData := make([]float64, len(rawCSVData))

// featureIndex and yIndex will track the current index of the matrix
values.
var featureIndex int
var yIndex int

// Sequentially move the rows into a slice of floats.
for idx, record := range rawCSVData {
    // Skip the header row.
    if idx == 0 {
        continue
    }

    // Loop over the float columns.
    for i, val := range record {
```

仅供非商业用途或交流学习使用


```
            // Convert the value to a float.
            valParsed, err := strconv.ParseFloat(val, 64)
            if err != nil {
                log.Fatal(err)
            }

            if i < 3 {

                // Add an intercept to the model.
                if i == 0 {
                    featureData[featureIndex] = 1
                    featureIndex++
                }

                // Add the float value to the slice of feature floats.
                featureData[featureIndex] = valParsed
                featureIndex++
            }

            if i == 3 {
                // Add the float value to the slice of y floats.
                yData[yIndex] = valParsed
                yIndex++
            }
        }
    }

    // Form the matrices that will be input to our regression.
    features := mat64.NewDense(len(rawCSVData), 4, featureData)
    y := mat64.NewVector(len(rawCSVData), yData)
```

接下来，用自变量矩阵和因变量矩阵创建一个新的 `ridge.RidgeRegression` 值，并调用 `Regress()` 方法来训练模型。然后可以打印出训练的回归公式：

```
    // Create a new RidgeRegression value, where 1.0 is the
    // penalty value.
    r := ridge.New(features, y, 1.0)

    // Train our regression model.
    r.Regress()

    // Print our regression formula.
    c1 := r.Coefficients.At(0, 0)
    c2 := r.Coefficients.At(1, 0)
    c3 := r.Coefficients.At(2, 0)
    c4 := r.Coefficients.At(3, 0)
    fmt.Printf("\nRegression formula:\n")
    fmt.Printf("y = %0.3f + %0.3f TV + %0.3f Radio + %0.3f Newspaper\n\n", c1,
    c2, c3, c4)
```

编译并运行该程序会得到以下回归公式：

```
$ go build
$ ./myprogram

Regression formula:
y = 3.038 + 0.047 TV + 0.177 Radio + 0.001 Newspaper
```

仅供非商业用途或交流学习使用

在这里，可以看到 TV 和 Radio 的系数与我们用最小二乘回归得到的系数相似，但它们略有不同。另外请注意，这里继续为 Newspaper 特性添加一个项。

可以通过创建 predict 函数来测试这个岭回归公式：

```
// predict uses our trained regression model to made a prediction based on
a
// TV, Radio, and Newspaper value.
func predict(tv, radio, newspaper float64) float64 {
        return 3.038 + tv*0.047 + 0.177*radio + 0.001*newspaper
}
```

然后，在测试示例中使用这个 predict 函数，来测试岭回归公式：

```
// Open the test dataset file.
f, err := os.Open("test.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()
// Create a new CSV reader reading from the opened file.
reader := csv.NewReader(f)

// Read in all of the CSV records
reader.FieldsPerRecord = 4
testData, err := reader.ReadAll()
if err != nil {
    log.Fatal(err)
}

// Loop over the holdout data predicting y and evaluating the prediction
// with the mean absolute error.
var mAE float64
for i, record := range testData {

    // Skip the header.
    if i == 0 {
        continue
    }

    // Parse the Sales.
    yObserved, err := strconv.ParseFloat(record[3], 64)
    if err != nil {
        log.Fatal(err)
    }

    // Parse the TV value.
    tvVal, err := strconv.ParseFloat(record[0], 64)
    if err != nil {
        log.Fatal(err)
    }

    // Parse the Radio value.
    radioVal, err := strconv.ParseFloat(record[1], 64)
    if err != nil {
        log.Fatal(err)
    }
```

仅供非商业用途或交流学习使用


```
        // Parse the Newspaper value.
        newspaperVal, err := strconv.ParseFloat(record[2], 64)
        if err != nil {
            log.Fatal(err)
        }

        // Predict y with our trained model.
        yPredicted := predict(tvVal, radioVal, newspaperVal)

        // Add the to the mean absolute error.
        mAE += math.Abs(yObserved-yPredicted) / float64(len(testData))
    }

    // Output the MAE to standard out.
    fmt.Printf("\nMAE = %0.2f\n\n", mAE)
```

编译并运行上述代码将得到如下新的 MAE 值：

```
$ go build
$ ./myprogram

MAE = 1.26
```

请注意，将 Newspaper 添加到模型并没有真正改善我们的 MAE。因此，在这种情况下并不是一个好主意，因为它增加了更多的复杂性，并且不会对模型性能提供任何显著的改变。

任何向模型增加的复杂度，都应该有添加的充分理由。理性地使用复杂模型将有助于处理棘手情况。

4.5 参考书目

线性回归：

- 普通最小二乘回归的可视化理解：http://setosa.io/ev/ordinary-least-squares-regression/
- github.com/sajari/regression 文档：http://godoc.org/github.com/sajari/regression

多元回归：

- 多元回归可视化：http://shiny.stat.calpoly.edu/3d_regression/

非线性和其他回归：

- go-hep.org/x/hep/fit 文档：https://godoc.org/go-hep.org/x/hep/fit
- github.com/berkmancenter/ridge 文档：https://godoc.org/github.com/berkmancenter/ridge

仅供非商业用途或交流学习使用

4.6 小结

恭喜！你已经使用 Go 正式完成了机器学习。具体来说，已经了解了回归模型，包括线性回归、多元回归、非线性回归和岭回归。应该能够在 Go 中实施基本线性回归和多元回归。

现在已经大致了解了机器学习，下一章将进一步讨论分类问题。

仅供非商业用途或交流学习使用

第 5 章
分　　类

当大多数人想到机器学习或者人工智能时，首先会想到利用机器学习解决分类问题。我们希望通过训练模型从有限分类中预测其中一种，用以解决这些问题。例如希望预测一个金融交易是否有欺诈性，或者希望预测一个图像中是否包含热狗、飞机、猫等。

尝试预测的类别可能从 2 种到成百上千种不等。除此之外，还可以依据几个或者多个属性来进行预测。所有这些组合产生的场景，可能形成大量具有不同假设、优点和缺点的模型。

在本章以及本书其他部分将涵盖某些模型，但为简单起见也会略过许多模型。然而，正如我们在本书中处理的任何问题一样，简单性和完整性应作为给用例选择模型类别的首要关注点。一些复杂和难懂的模型可能能很好地解决特定问题，但对于这里的大部分情况，并不需涉及这些模型。使用简单和可解释的分类模型仍然是我们的目标之一。

5.1　理解分类模型的术语

同回归一样，分类问题也有一些自己的术语。虽然与回归中的术语有部分重叠，但分类也有自己独特的部分。

- **类别**、**标签**或者**种类**：这些术语都可用作表示预测的各种不同选择。例如，可以有一个欺诈类和一个非欺诈类，或者可以有坐着、站着、跑动和步行类别。
- **二元分类**：这种分类是从两个种类或类别选取一种，例如是 / 否或欺诈 / 非

仅供非商业用途或交流学习使用

欺诈。

- **多元分类**：这种分类是从两个以上类别中选取一种。例如尝试分类一个图像为热狗、飞机、猫等。
- **标记数据**或**注释数据**：真实世界的观察记录，或者相应类别的配对记录。例如，如果通过交易时间预测欺诈性，这个数据会包括一系列测量的交易时间，以及对应的标签以表明它们是否具有欺诈性。

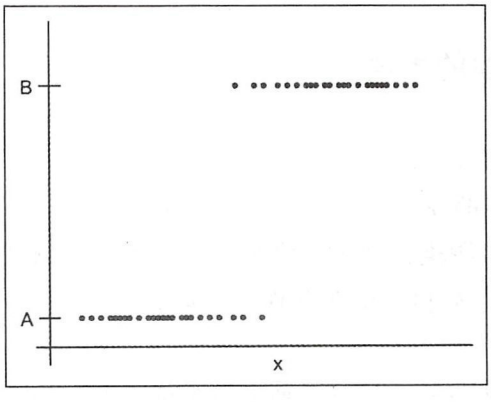
5.2 逻辑回归

第一个要谈论的模型称为**逻辑回归**（logistic regression）。从名称上可以看出，这个方法是基于在前一章已经详细讨论过的回归算法。但是这种特定的回归所使用的函数，特别适合分类问题。

这同样是一个简单和可解释的模型，这使其成为解决分类问题的首选。有许多现有的 Go 软件包可以实现逻辑回归，其中包括 `github.com/xlvector/hector`、`github.com/cdipaolo/goml` 以及 `github.com/sjwhitworth/golearn`。然而在本例中，将从零开始实施逻辑回归，这样既可以全面理解模型的训练，又可以理解逻辑回归的简单性。此外在某些情况下，可能希望像下文那样从头做起，这样可以避免代码库中的额外依赖。

5.2.1 逻辑回归概述

假设尝试预测两个类别，A 和 B。再假设试图基于变量 x 预测 A 或 B。类别 A 和 B 可能看起来如下所示：



虽然可以画一条线段来模拟这种行为，但这显然不是线性行为，也不符合线性

回归的假设。作为 x 的函数，数据的形状更像是从一个类别到另一个类别的过程。这里真正需要的函数是在 x 值较小时留在 A 处，x 值较大时留在 B 处。

幸运的是有这样的函数。该函数称为**逻辑函数**（logistic function）。这也是逻辑回归名称的来历，形式如下：

$$f(x)=\frac{1}{1+e^{-x}}$$

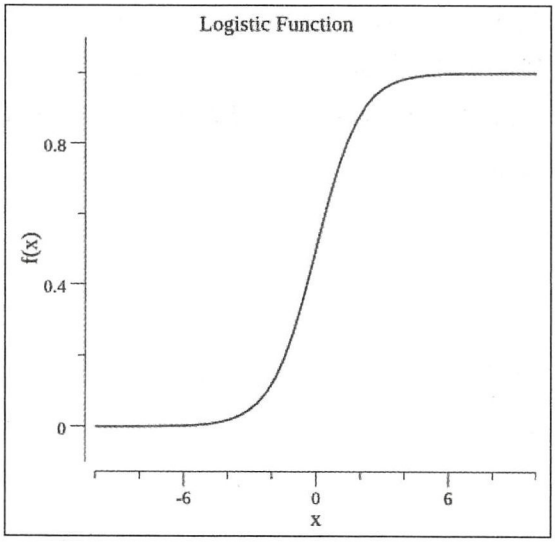
以下是 Go 语言的实现：

```
// logistic implements the logistic function, which
// is used in logistic regression.
func logistic(x float64) float64 {
        return 1 / (1 + math.Exp(-x))
}
```

用 gonum.org/v1/plot 来绘制逻辑函数，看看是什么样子。

```
// Create a new plot.
p, err := plot.New()
if err != nil {
    log.Fatal(err)
}
p.Title.Text = "Logistic Function"
p.X.Label.Text = "x"
p.Y.Label.Text = "f(x)"

// Create the plotter function.
logisticPlotter := plotter.NewFunction(func(x float64) float64 { return
logistic(x) })
logisticPlotter.Color = color.RGBA{B: 255, A: 255}

// Add the plotter function to the plot.
p.Add(logisticPlotter)

// Set the axis ranges.  Unlike other data sets,
// functions don't set the axis ranges automatically
// since functions don't necessarily have a
// finite range of x and y values.
p.X.Min = -10
p.X.Max = 10
p.Y.Min = -0.1
p.Y.Max = 1.1

// Save the plot to a PNG file.
if err := p.Save(4*vg.Inch, 4*vg.Inch, "logistic.png"); err != nil {
    log.Fatal(err)
}
```

编译和运行绘制代码，构建如下的图形：



如你所见，这个函数有着我们正在寻找的阶梯状的行为，用于构建类别 *A* 和 *B* 的阶梯（想象 *A* 对应于 0.0，*B* 对应于 1.0）。

不仅如此，这个逻辑函数还有一些非常便利的、在分类时可以利用的属性。为了看到这一点，让我们退一步考虑如何创建模型 p，用以表示类别 *A* 或 *B* 发生的概率。一种方法是**对比值比**（odds ratio）取 log，构建如 $\log(p/(1-p))$ 的线性模型。通过比值比可以了解，分类 *A* 的存在与否如何影响到分类 *B* 存在与否。使用这个奇怪的 log（被称为**对数**）的意义将很快看到，但现在，假设想要按如下方式线性建模：

$$\log\left(\frac{p}{1-p}\right)=mx+b$$

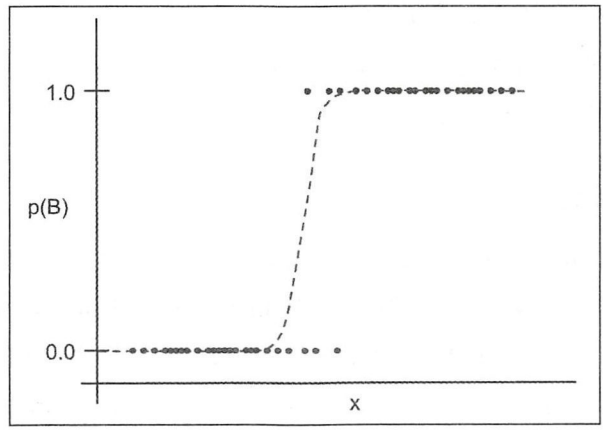
现在，对比值比应用指数会得到如下结果：

$$\frac{p}{1-p}=e^{mx+b}$$

简化前面方程式后，得到需要的结果：

$$p=\frac{p}{1+e^{-mx-b}}$$

如果看看这个等式的右半部分，可以发现逻辑函数已经出现了。等式为我们的假设提供了正式基础，逻辑函数擅长于构建 *A*、*B* 两类的分类函数。例如，如果把 p 作为 *B* 的观察概率并拟合逻辑函数到数据中，便能得到以 x 为变量的函数 *B* 的概率预测模型（从而 1 减去 *A* 的预测概率）。如下图所示，已经重新整理了之前的 *A*、*B* 图表，并覆盖了模拟概率的逻辑函数。



因此，创建一个逻辑回归模型包含找到最大化观察数量的、可用于预测的逻辑函数。

请注意逻辑回归的优点之一是仍保持简单和可解释。然而，其中系数 m 和 b 跟线性回归的理解有些不同。系数 m（如有多个自变量，可用系数 m_1，m_2 表示）同比值比呈指数关系。因此，如果系数 m 是 0.5，那么可以通过 $exp(0.5\ x)$ 同比值比关联。如果有两个系数 $exp(0.5\ x_1 + 1.0\ x_2)$，那么可以得出结论，模型类对于 x_1 的比值是 $exp(0.5)=1.65$，对于 x_2 的比值是 $exp(1.0)=2.72$，换句话说，不能直接比较系数，而需要放到指数中。

5.2.2　逻辑回归的假设和陷阱

还记得线性回归中很长的假设列表吗？逻辑回归不会受制于那些相同的假设。然而在使用逻辑回归时，仍然存在一些重要的假设：

- **比值的对数呈线性关系**：正如前面提到的，逻辑回归潜在的假设是可以构建比值比对数的线性模型。
- **因变量的编码**：在前面创建的模型中，假设尝试预测 B 的概率，其中 1.0 的概率对应示例 B。因此也需要按这种编码类型准备数据。稍后会在下面示例中演示。
- **观察的独立性**：数据中每个 x 示例都应该保持独立，也就是说，要避免相同示例被包含多次。

同样，需要留意如下的一些逻辑回归常见陷阱：

- 与其他分类技术相比，逻辑回归对异常值更加敏感。应留心这点并尝试相应地分析数据。
- 由于逻辑回归所依赖的指数函数从未真正达到 0.0 或者 1.0（除了 +/– 无穷大

仅供非商业用途或交流学习使用

以外），因此评估指标不太可能退化。

大家公认，逻辑回归是保持了可解释性的非常强大的方法。它是一个灵活的模型，在解决分类问题时应当首先考虑。

5.2.3 逻辑回归示例

这里将使用LendingClub贷款问题的数据作为数据集，以说明逻辑回归。LendingClub会每季度发布一次数据，从https://www.lendingclub.com/info/download-data.action可以获得。这里将使用这份数据精简版本（可以在本书附带的代码包中找到），其中仅包括两列，`FICO.Range`（代表由Fair, Isaac and Company或者FICO提供的贷款申请人信用评分）和`Interest.Rate`（代表给予贷款申请人的贷款利率）。数据如下所示：

```
$ head loan_data.csv
FICO.Range,Interest.Rate
735-739,8.90%
715-719,12.12%
690-694,21.98%
695-699,9.99%
695-699,11.71%
670-674,15.31%
720-724,7.90%
705-709,17.14%
685-689,14.33%
```

本练习目标是创建一个逻辑回归模型，它将告诉我们，对于给定的信用评分，是否可以获得等于或低于特定利率的贷款。例如，假设希望获得低于12%的利率。模型可以告诉我们对于给定的信用评分，能（是，或者第一类）或者不能（否，或者第二类）获得贷款。

5.2.3.1 清洗和分析数据

回顾前面的贷款数据样本，可以看到同所需要的数据形式并不完全一致。具体来说，需要做以下工作：

1. 从利率和FICO评分列中删除非数字字符。
2. 根据给定阈值，将利率编码成两类。这里使用1.0代表第一类（是，可以获得该利率贷款），0.0代表第二类（否，不能获得该利率贷款）。
3. 选择一个数值作为FICO信用评分。我们已经知道信用评分范围，但是这里需要一个单一数值。评分的均值、最小值、最大值都是很自然的选择。在本例中将使用最小值（为了保守）。
4. 在这里，将采用**标准化**FICO评分（通过每个评分减去最小评分，然后除以评分范围）的方法。这可以将数值分散在0.0和1.0之间。这么做需要一个理

仅供非商业用途或交流学习使用

由，因为这使得数据变得难读。但是，确实有个很好的理由。这里将使用梯度下降法来训练逻辑回归，该方法使用规范的数据能获得更好性能。实际上，当使用非标准数据运行相同示例时，会出现收敛问题。

下面来编写 Go 程序用给定利率（本例使用 12%）清洗数据。从给定文件中读取数据，使用 encoding/csv 解析数据，然后将清洗后的数据输出到 clean_loan_data.csv 文件中。在数据清洗中，将使用如下定义为常量的最小值和最大值。

```
const (
    scoreMax = 830.0
    scoreMin = 640.0
)
```

然后，实际的清洗函数如下所示：

```
// Open the loan dataset file.
f, err := os.Open("loan_data.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a new CSV reader reading from the opened file.
reader := csv.NewReader(f)
reader.FieldsPerRecord = 2

// Read in all of the CSV records
rawCSVData, err := reader.ReadAll()
if err != nil {
    log.Fatal(err)
}

// Create the output file.
f, err = os.Create("clean_loan_data.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a CSV writer.
w := csv.NewWriter(f)

// Sequentially move the rows writing out the parsed values.
for idx, record := range rawCSVData {

    // Skip the header row.
    if idx == 0 {

        // Write the header to the output file.
        if err := w.Write(record); err != nil {
            log.Fatal(err)
        }
        continue
    }
```

仅供非商业用途或交流学习使用

```
        // Initialize a slice to hold our parsed values.
        outRecord := make([]string, 2)

        // Parse and standardize the FICO score.
        score, err := strconv.ParseFloat(strings.Split(record[0], "-")[0], 64)
        if err != nil {
            log.Fatal(err)
        }

        outRecord[0] = strconv.FormatFloat((score-scoreMin)/(scoreMax-
scoreMin), 'f', 4, 64)

        // Parse the Interest rate class.
        rate, err := strconv.ParseFloat(strings.TrimSuffix(record[1], "%"), 64)
        if err != nil {
            log.Fatal(err)
        }

        if rate <= 12.0 {
            outRecord[1] = "1.0"

            // Write the record to the output file.
            if err := w.Write(outRecord); err != nil {
                log.Fatal(err)
            }
            continue
        }

        outRecord[1] = "0.0"

        // Write the record to the output file.
        if err := w.Write(outRecord); err != nil {
            log.Fatal(err)
        }
    }

    // Write any buffered data to the underlying writer (standard output).
    w.Flush()

    if err := w.Error(); err != nil {
        log.Fatal(err)
    }
```

编译并运行程序可得到所需的输出：

```
$ go build
$ ./example3
$ head clean_loan_data.csv
FICO_score,class
0.5000,1.0
0.3947,0.0
0.2632,0.0
0.2895,1.0
0.2895,1.0
0.1579,0.0
0.4211,1.0
0.3421,0.0
0.2368,0.0
```

仅供非商业用途或交流学习使用

非常好！得到了所需格式的数据。现在为FICO评分和利率数据创建直方图并计算汇总统计，用以获得对数据更直观的认识。这里将使用github.com/kniren/gota/dataframe生成汇总统计，使用gonum.org/v1/plot生成直方图。

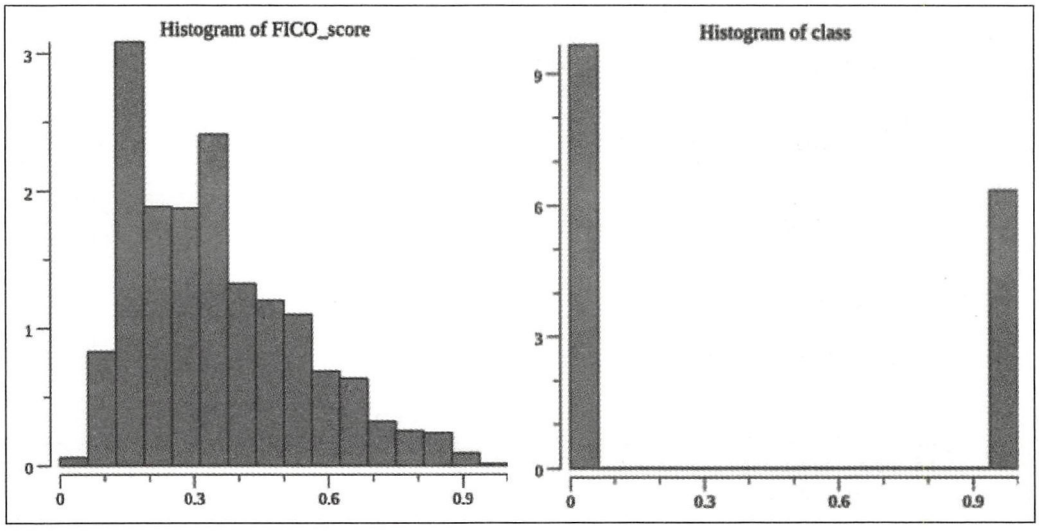
```
// Open the CSV file.
loanDataFile, err := os.Open("clean_loan_data.csv")
if err != nil {
    log.Fatal(err)
}
defer loanDataFile.Close()

// Create a dataframe from the CSV file.
loanDF := dataframe.ReadCSV(loanDataFile)

// Use the Describe method to calculate summary statistics
// for all of the columns in one shot.
loanSummary := loanDF.Describe()

// Output the summary statistics to stdout.
fmt.Println(loanSummary)

// Create a histogram for each of the columns in the dataset.
for _, colName := range loanDF.Names() {

    // Create a plotter.Values value and fill it with the
    // values from the respective column of the dataframe.
    plotVals := make(plotter.Values, loanDF.Nrow())
    for i, floatVal := range loanDF.Col(colName).Float() {
        plotVals[i] = floatVal
    }

    // Make a plot and set its title.
    p, err := plot.New()
    if err != nil {
        log.Fatal(err)
    }
    p.Title.Text = fmt.Sprintf("Histogram of a %s", colName)

    // Create a histogram of our values.
    h, err := plotter.NewHist(plotVals, 16)
    if err != nil {
        log.Fatal(err)
    }

    // Normalize the histogram.
    h.Normalize(1)

    // Add the histogram to the plot.
    p.Add(h)

    // Save the plot to a PNG file.
    if err := p.Save(4*vg.Inch, 4*vg.Inch, colName+"_hist.png"); err != nil
{
        log.Fatal(err)
    }
}
```

仅供非商业用途或交流学习使用

运行上述代码，得到以下输出：

```
$ go build
$ ./myprogram
[7x3] DataFrame

    column FICO_score class
 0: mean 0.346782 0.396800
 1: stddev 0.184383 0.489332
 2: min 0.000000 0.000000
 3: 25% 0.210500 0.000000
 4: 50% 0.315800 0.000000
 5: 75% 0.447400 1.000000
 6: max 1.000000 1.000000
    <string> <float> <float>

$ ls *.png
class_hist.png FICO_score_hist.png
```

可以看到平均信用评分是相当高的 706.1，并且在类别 1 和类别 0 之间的平衡也非常好，表明平均值是 0.5 左右。但是，类别 0 的例子似乎更多（代表没有收到利率为 12% 或者更低的贷款）。此外，`*.png` 直方图如下所示：



这证实了我们对类别之间平衡的怀疑，并表明 FICO 评分略微偏向低位值。

5.2.3.2 创建训练和测试集

与前一章示例类似，需要将数据分割为训练集和测试集。将再次使用 `github.com/kniren/gota/dataframe` 来执行此操作：

```
// Open the clean loan dataset file.
f, err := os.Open("clean_loan_data.csv")
if err != nil {
    log.Fatal(err)
}
```

仅供非商业用途或交流学习使用

```
defer f.Close()
// Create a dataframe from the CSV file.
// The types of the columns will be inferred.
loanDF := dataframe.ReadCSV(f)

// Calculate the number of elements in each set.
trainingNum := (4 * loanDF.Nrow()) / 5
testNum := loanDF.Nrow() / 5
if trainingNum+testNum < loanDF.Nrow() {
    trainingNum++
}

// Create the subset indices.
trainingIdx := make([]int, trainingNum)
testIdx := make([]int, testNum)

// Enumerate the training indices.
for i := 0; i < trainingNum; i++ {
    trainingIdx[i] = i
}

// Enumerate the test indices.
for i := 0; i < testNum; i++ {
    testIdx[i] = trainingNum + i
}

// Create the subset dataframes.
trainingDF := loanDF.Subset(trainingIdx)
testDF := loanDF.Subset(testIdx)

// Create a map that will be used in writing the data
// to files.
setMap := map[int]dataframe.DataFrame{
    0: trainingDF,
    1: testDF,
}

// Create the respective files.
for idx, setName := range []string{"training.csv", "test.csv"} {

    // Save the filtered dataset file.
    f, err := os.Create(setName)
    if err != nil {
        log.Fatal(err)
    }

    // Create a buffered writer.
    w := bufio.NewWriter(f)

    // Write the dataframe out as a CSV.
    if err := setMap[idx].WriteCSV(w); err != nil {
        log.Fatal(err)
    }
}
```

编译并运行程序将生成测试和训练的两个示例文件：

仅供非商业用途或交流学习使用

```
$ go build
$ ./myprogram
$ wc -l *.csv
 2046 clean_loan_data.csv
  410 test.csv
 1638 training.csv
 4094 total
```

5.2.3.3 训练和测试逻辑回归模型

现在，创建一个用于训练逻辑回归模型的函数。这个函数需要执行以下操作：

1. 接受 FICO 评分数据作为自变量。
2. 添加截距到模型中。
3. 初始化并优化逻辑回归模型的系数（或权重）。
4. 返回定义训练模型的最优权重。

为了优化系数 / 权重，将使用称为**随机梯度下降**（stochastic gradient descent）的技术。这项技术在附录中有更详细的介绍。现在，只需了解正在尝试使用一些非优化权重进行预测，并计算这些权重的误差，然后迭代更新它们使得正确预测的概率最大化。

这个优化的实现如下，该函数使用以下输入：

- `features`：一个指向 `gonum mat64.Dense` 矩阵的指针。该矩阵包含正在使用的任何自变量（在本例中是 FICO 评分）的列，以及代表截距 1.0 的列。
- `labels`：包含同 `features` 对应的所有类标签的浮点数切片。
- `numSteps`：优化的最大迭代次数
- `LearningRate`：一个可调节的参数，有助于优化收敛。

该函数输出逻辑回归模型的优化权重：

```
// logisticRegression fits a logistic regression model
// for the given data.
func logisticRegression(features *mat64.Dense, labels []float64, numSteps
int, learningRate float64) []float64 {

        // Initialize random weights.
        _, numWeights := features.Dims()
        weights := make([]float64, numWeights)

        s := rand.NewSource(time.Now().UnixNano())
        r := rand.New(s)

        for idx, _ := range weights {
                weights[idx] = r.Float64()
        }

        // Iteratively optimize the weights.
        for i := 0; i < numSteps; i++ {

        // Initialize a variable to accumulate error for this iteration.
        var sumError float64
```

仅供非商业用途或交流学习使用

```
        // Make predictions for each label and accumulate error.
        for idx, label := range labels {

            // Get the features corresponding to this label.
            featureRow := mat64.Row(nil, idx, features)

            // Calculate the error for this iteration's weights.
            pred := logistic(featureRow[0]*weights[0]
            featureRow[1]*weights[1])
            predError := label - pred
            sumError += math.Pow(predError, 2)

            // Update the feature weights.
            for j := 0; j < len(featureRow); j++ {
                weights[j] += learningRate * predError * pred * (1 - pred)
* featureRow[j]
            }
        }
    }

    return weights
}
```

如你所见，这个函数相对紧凑和简单。这有助于保持代码的可读性，团队成员可以迅速了解模型中发生的情况，而不是将其隐藏在黑盒子之中。

尽管 R 和 Python 在机器学习领域非常流行，但可以看到，使用 Go 语言可以快速而紧凑地实现机器学习算法。而且这些实现能迅速达到完备水平，远远超过其他语言的原始实现。

为了在训练数据集上训练逻辑回归模型，将使用 encoding/csv 解析训练文件，然后提供必要的参数到 logisticRegression 函数中。该过程如下所示，包含一些用于输出训练的逻辑公式到 stdout 中的代码。

```
// Open the training dataset file.
f, err := os.Open("training.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a new CSV reader reading from the opened file.
reader := csv.NewReader(f)
reader.FieldsPerRecord = 2

// Read in all of the CSV records
rawCSVData, err := reader.ReadAll()
if err != nil {
    log.Fatal(err)
}

// featureData and labels will hold all the float values that
// will eventually be used in our training.
```

仅供非商业用途或交流学习使用


```
featureData := make([]float64, 2*len(rawCSVData))
labels := make([]float64, len(rawCSVData))

// featureIndex will track the current index of the features
// matrix values.
var featureIndex int

// Sequentially move the rows into the slices of floats.
for idx, record := range rawCSVData {

    // Skip the header row.
    if idx == 0 {
        continue
    }

    // Add the FICO score feature.
    featureVal, err := strconv.ParseFloat(record[0], 64)
    if err != nil {
        log.Fatal(err)
    }

    featureData[featureIndex] = featureVal

    // Add an intercept.
    featureData[featureIndex+1] = 1.0

    // Increment our feature row.
    featureIndex += 2

    // Add the class label.
    labelVal, err := strconv.ParseFloat(record[1], 64)
    if err != nil {
        log.Fatal(err)
    }

    labels[idx] = labelVal
}

// Form a matrix from the features.
features := mat64.NewDense(len(rawCSVData), 2, featureData)

// Train the logistic regression model.
weights := logisticRegression(features, labels, 100, 0.3)

// Output the Logistic Regression model formula to stdout.
formula := "p = 1 / ( 1 + exp(- m1 * FICO.score - m2) )"
fmt.Printf("\n%s\n\nm1 = %0.2f\nm2 = %0.2f\n\n", formula, weights[0],
weights[1])
```

编译并运行此训练功能会生成以下经过训练的逻辑回归公式：

```
$ go build
$ ./myprogram

p = 1 / ( 1 + exp(- m1 * FICO.score - m2) )

m1 = 13.65
m2 = -4.89
```

仅供非商业用途或交流学习使用

然后可以直接利用这个公式来做预测。但请记住，该模型预测了获得贷款的可能性（利率为12%）。因此，需要在进行预测时使用概率阈值。例如，可以说任何0.5+的 p 将被视为正数（第一类，或获得贷款），任何较低的 p 值将被视为负数。这种类型的预测在以下函数中实现：

```
// predict makes a prediction based on our
// trained logistic regression model.
func predict(score float64) float64 {

    // Calculate the predicted probability.
    p := 1 / (1 + math.Exp(-13.65*score+4.89))

    // Output the corresponding class.
    if p >= 0.5 {
        return 1.0
    }

    return 0.0
}
```

利用这个 `predict` 函数，可以使用本书前面介绍的一个评估指标来评估训练的逻辑回归模型。这里将使用准确度作为评估指标，如下面的代码所示：

```
// Open the test examples.
f, err := os.Open("test.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a new CSV reader reading from the opened file.
reader := csv.NewReader(f)

// observed and predicted will hold the parsed observed and predicted
values
// form the labeled data file.
var observed []float64
var predicted []float64

// line will track row numbers for logging.
line := 1

// Read in the records looking for unexpected types in the columns.
for {

    // Read in a row. Check if we are at the end of the file.
    record, err := reader.Read()
    if err == io.EOF {
        break
    }

    // Skip the header.
    if line == 1 {
        line++
```

仅供非商业用途或交流学习使用


```
            continue
        }
        // Read in the observed value.
        observedVal, err := strconv.ParseFloat(record[1], 64)
        if err != nil {
            log.Printf("Parsing line %d failed, unexpected type\n", line)
            continue
        }

        // Make the corresponding prediction.
        score, err := strconv.ParseFloat(record[0], 64)
        if err != nil {
            log.Printf("Parsing line %d failed, unexpected type\n", line)
            continue
        }

        predictedVal := predict(score)

        // Append the record to our slice, if it has the expected type.
        observed = append(observed, observedVal)
        predicted = append(predicted, predictedVal)
        line++
    }

    // This variable will hold our count of true positive and
    // true negative values.
    var truePosNeg int

    // Accumulate the true positive/negative count.
    for idx, oVal := range observed {
        if oVal == predicted[idx] {
            truePosNeg++
        }
    }

    // Calculate the accuracy (subset accuracy).
    accuracy := float64(truePosNeg) / float64(len(observed))

    // Output the Accuracy value to standard out.
    fmt.Printf("\nAccuracy = %0.2f\n\n", accuracy)
```

对数据运行本测试可以得到如下准确度：

```
$ go build
$ ./myprogram

Accuracy = 0.83
```

太好了！对于用大约 30 行 Go 语言实现的机器学习模型，83% 的准确度并不是那么糟糕。通过这个简单的模型能够预测，给定某个信用评分，贷款申请人的贷款利率是否可以少于或等于 12%。不仅如此，我们是用来自真实公司的实际的杂乱数据完成的。

仅供非商业用途或交流学习使用

5.3　k- 最近邻

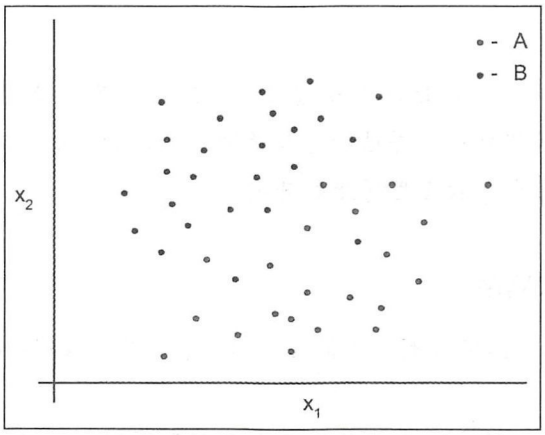
离开逻辑回归，让我们试试第一个非回归模型，即 **k- 最近邻**（kNN）。kNN 也是一个简单的分类模型，它是最简单的模型算法之一。该算法源自一个基本前提：如果想对一个记录分类，同时需要考虑其他类似记录。

kNN 在多个 Go 软件包中有实现，包括 `github.com/sjwhitworth/golearn`、`github.com/rikonor/go-ann`、`github.com/akreal/knn` 和 `github.com/cdipaolo/goml`。本书将使用 `github.com/sjwhitworth/golearn` 实现，这样可以很好地介绍 `github.com/sjwhitworth/golearn` 的使用。

5.3.1　kNN 概述

如前所述，kNN 的运作原则是根据类似的记录对记录进行分类。在定义相似方面，还有很多细节需要处理。但是同其他模型相比，KNN 不会有复杂的参数和选项。

再想象一下，有两个分类 *A* 和 *B*。但是这一次，假设要根据两个特征 x_1 和 x_2 进行分类。从外观上，看起来如下所示：

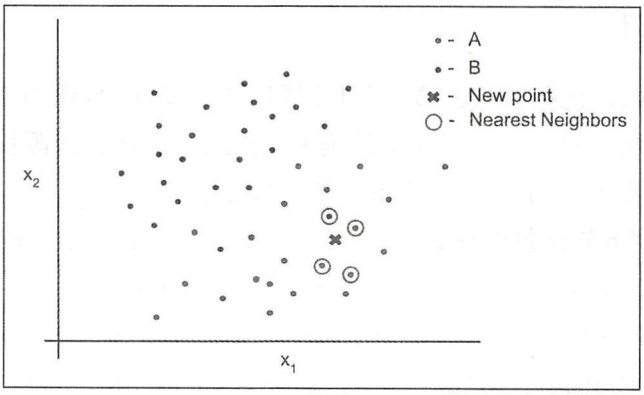


现在，假设有一个未知类别的新数据点。这个新的数据点将坐落在这个空间的某个地方。根据 kNN 所描述，为了分类这个新点，应该执行以下操作：

1. 根据一些近似度量（例如，在 x_1 和 x_2 的这个空间中的直线距离），找到距离新点的 k 个最近点。
2. 确定这 *k* 个最近的邻居中有多少是 *A* 类的，有多少是 *B* 类的。
3. 将这个新点分类为 *k* 个最近邻居中的主导类。

例如，如果选择将 *k* 值设为 4，这将看起来如下所示：

仅供非商业用途或交流学习使用



这个神秘点有三个最近的 *A* 邻居和一个最近的 *B* 邻居。因此，会将新的神秘点归类为 *A* 类点。

有很多相似性度量可以用来确定 *k* 个最近邻。其中最常见的就是欧几里得距离，它就是由你的特征（在我们的例子中是 x_1 和 x_2）所构成的空间中的一点到另一点的直线距离。其他还包括**曼哈顿距离**（Manhattan distance）、**Minkowski 距离**（Minkowski distance）、**余弦相似度**（cosine similarity）和 **Jaccard 相似度**（Jaccard Similarity）。

与评估指标一样，测量距离或相似性的方法也很多。使用 kNN 时，应该考虑这些测量方法的优缺点，并选择适合你的用例和数据的方法。但是如果还有疑问的话，可以考虑从欧几里得距离开始进行。

5.3.2 kNN 假设和陷阱

由于其原理简单，kNN 没有太多假设。但是，在应用 kNN 时应该注意一些常见的问题：

- kNN 属于惰性评估。这意味着只有当需要预测时，才计算距离或相似度。在预测之前，并不训练或匹配任何东西。这样做有它的优势，但是当有很多数据点时，计算和搜索点时可能会很慢。
- *k* 是可以随意选择的，但在选择 *k* 时也有一些形式化的方法，用以合理选择 *k*。选择 *k* 的常用技术就是搜索 *k* 值的范围。例如，可以从 *k=2* 开始。然后，可以开始增加 *k*，并且对于每个 *k*，在一个测试集上进行评估。
- kNN 没有考虑哪些特征相对于其他特征更重要。而且，如果一个特征的规模比其他特征大得多，这可能会不自然地加重这些较大特征的重要性。

仅供非商业用途或交流学习使用

5.3.3 kNN 示例

在这里，以及本章其他示例中，将使用关于鸢尾花的数据集来解决经典的分类问题。数据集如下所示：

```
$ head iris.csv
sepal_length,sepal_width,petal_length,petal_width,species
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
5.0,3.6,1.4,0.2,Iris-setosa
5.4,3.9,1.7,0.4,Iris-setosa
4.6,3.4,1.4,0.3,Iris-setosa
5.0,3.4,1.5,0.2,Iris-setosa
4.4,2.9,1.4,0.2,Iris-setosa
```

前四列是鸢尾花的各种测量值，最后一列是相应的物种标签。这个例子的目标是创建一个 kNN 分类器，该分类器能够从一组测量结果中预测鸢尾花的种类。有三种花，或三类，使得这是一个多级分类（相对于逻辑回归进行的二元分类）。

你可能还记得在第 2 章已经详细描述了鸢尾花数据集，因此这里不会重新整理数据。然而，在开发 kNN 模型时，直观了解数据仍然很重要。回到第 2 章，以便回顾这个数据集中变量的分布。

在这个例子中，将使用 github.com/sjwhitworth/golearn。在 github.com/sjwhitworth/golearn 中实现了各种机器学习的模型，包括 kNN 和其他一些即将探讨的模型。github.com/sjwhitworth/golearn 也实现了交叉验证。这里将利用交叉验证方便、能避免手工切割训练集和测试集的优点，来训练、测试和验证。

要使用任何 github.com/sjwhitworth/golearn 中的模型，必须首先将数据转化为称作 instances 的 github.com/sjwhitworth/golearn 内部格式。对于鸢尾花数据，可以如下操作：

```
// Read in the iris data set into golearn "instances".
irisData, err := base.ParseCSVToInstances("iris.csv", true)
if err != nil {
    log.Fatal(err)
}
```

然后，初始化 kNN 模型并进行交叉验证，这一步非常简单快捷：

```
// Initialize a new KNN classifier.  We will use a simple
// Euclidean distance measure and k=2.
knn := knn.NewKnnClassifier("euclidean", "linear", 2)
```

仅供非商业用途或交流学习使用


```
// Use cross-fold validation to successively train and evaluate the model
// on 5 folds of the data set.
cv, err :=
evaluation.GenerateCrossFoldValidationConfusionMatrices(irisData, knn, 5)
if err != nil {
    log.Fatal(err)
}
```

最后，可以得到5折交叉验证的平均准确度，并将准确度输出到stdout：

```
// Get the mean, variance and standard deviation of the accuracy for the
// cross validation.
mean, variance := evaluation.GetCrossValidatedMetric(cv,
evaluation.GetAccuracy)
stdev := math.Sqrt(variance)

// Output the cross metrics to standard out.
fmt.Printf("\nAccuracy\n%.2f (+/- %.2f)\n\n", mean, stdev*2)
```

编译和运行这些内容，得到以下输出：

```
$ go build
$ ./myprogram
Optimisations are switched off
Optimisations are switched off
Optimisations are switched off
Optimisations are switched off
Optimisations are switched off
KNN: 95.00 % done
Accuracy
0.95 (+/- 0.05)
```

在交叉验证输出一些日志后，可以看到kNN（k=2）能够以95%的准确度预测鸢尾花的物种！

下一步就是用各种不同的k值试用这个模型。实际上，将精确度与k值进行比较以查看哪个k值可以提供最佳性能是一个很好的练习。

5.4 决策树和随机森林

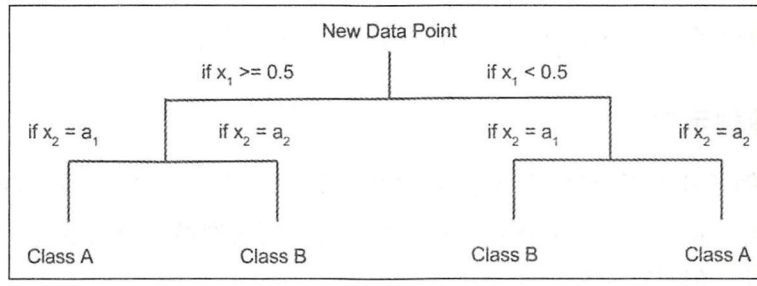
基于树的模型与之前已经讨论过的其他类型的模型有很大不同，但它们被广泛使用并且非常强大。如果有一系列的if-then语句应用在数据中，那么可以考虑使用**决策树**模型。训练这类模型的过程，就是在构建一系列的控制流语句，用于识别记录。

决策树在github.com/sjwhitworth/golearn和github.com/xlvector/hector等中有实现，随机森林在github.com/sjwhitworth/golearn、github.com/xlvector/hector和github.com/ryanbressler/CloudForest中有实现。下面的示例中将再次使用github.com/sjwhitworth/golearn。

仅供非商业用途或交流学习使用

5.4.1 决策树和随机森林概述

再次考虑类别 A 和类别 B。在这种情况下，假设有一个特征 x_1，范围从 0.0 到 1.0，而我们有另一个特征 x_2，它是分类的，取值是 a_1 和 a_2 其中一个（这可能是像男性 / 女性或红 / 蓝）。用于分类新数据点的决策树可能如下所示：



有多种方法可以选择决策树的构建方式、拆分方式等。确定决策树如何构建的最常见方法之一是使用称为**熵**（entropy）的量。这个基于熵的方法在附录中有更详细的讨论，但基本上，创建和分割树是基于那些可以为待解问题提供最多信息的特征。特征越重要，在树上的优先级就越高。

重要特征的优先级和清晰的结构使得决策树非常易于理解。也使得决策树对于那些需要解释推断（例如合规性要求）的应用来说非常重要。

但是，对于训练数据的变化，单个决策树可能不稳定。换句话说，即使训练数据发生微小变化，树的结构也可能发生显著变化。这会在运作管理和认知管理方面都具有挑战性，这也是随机森林模型创建的原因之一。

随机森林是决策树的集合，它们共同作出预测。与单一决策树相比，随机森林更加稳定，并且它们对于过度拟合更为稳健。事实上，这种将模型整合到一个**整体**中的想法在整个机器学习中非常普遍，既可以提高简单分类器（如决策树）的性能，又可以防止过度拟合。

为了构造一个随机森林，可以选择特征的 N 个随机子集，并基于这些子集构造 N 个独立的决策树。在进行预测时，让这 N 个决策树中的每一个都进行预测。为了获得最终的预测，可以对这 N 个预测进行多数投票。

5.4.2 决策树和随机森林的假设及陷阱

基于树的方法是非统计方法，没有许多与回归相关的假设。但是，请记住一些陷阱：

- 单一的决策树模型容易过度拟合数据，特别是在不限制树的深度的情况下。大多数实现允许通过一个参数来限制这个深度（或者**修剪**决策树）。修剪参数

仅供非商业用途或交流学习使用

通常会允许删除对预测影响不大的部分树，从而降低模型的总体复杂度。

- 当开始谈论诸如随机森林之类的集合模型时，我们正在进入会有些不透明的模型。很难获得一个模型的整体直觉，必须在某种程度上像黑盒子一样对待它。因此只有在必要时才应用这些较难解释的模型。
- 虽然决策树本身的计算效率非常高，但随机森林的计算效率可能非常低，具体取决于拥有多少特征以及随机森林中有多少棵树。

5.4.3 决策树示例

在这个例子中再次使用鸢尾花数据集。你已经学会了如何在 github.com/sjwhitworth/golearn 中处理这个数据集，可以再次遵循类似的模式。这里将再次使用交叉验证，但是这次拟合的是决策树模型：

```
// Read in the iris data set into golearn "instances".
irisData, err := base.ParseCSVToInstances("iris.csv", true)
if err != nil {
    log.Fatal(err)
}

// This is to seed the random processes involved in building the
// decision tree.
rand.Seed(44111342)

// We will use the ID3 algorithm to build our decision tree.  Also, we
// will start with a parameter of 0.6 that controls the train-prune split.
tree := trees.NewID3DecisionTree(0.6)

// Use cross-fold validation to successively train and evaluate the model
// on 5 folds of the data set.
cv, err :=
evaluation.GenerateCrossFoldValidationConfusionMatrices(irisData, tree, 5)
if err != nil {
    log.Fatal(err)
}

// Get the mean, variance and standard deviation of the accuracy for the
// cross validation.
mean, variance := evaluation.GetCrossValidatedMetric(cv,
evaluation.GetAccuracy)
stdev := math.Sqrt(variance)

// Output the cross metrics to standard out.
fmt.Printf("\nAccuracy\n%.2f (+/- %.2f)\n\n", mean, stdev*2)
```

编译和运行该决策树模型得到以下内容：

```
$ go build
$ ./myprogram

Accuracy
0.94 (+/- 0.06)
```

这次的准确率为94%。略逊于之前的kNN模型，但准确率已经很高了。

5.4.4 随机森林的例子

github.com/sjwhitworth/golearn也实现了随机森林。为了利用随机森林解决鸢尾花问题，这里简单地将决策树模型换成随机森林。需要告诉软件包要构建多少棵树以及每棵树有多少个随机选择的特征。

每棵树特征数量的理想默认值是总特征数的平方根，在本例中设置为2。我们将看到，对于小数据集，这种选择并不会产生很好的结果，因为做一个好的预测需要用到所有的特征值。然而这里将使用默认的配置来说明随机森林，以了解它的工作原理：

```
// Assemble a random forest with 10 trees and 2 features per tree,
// which is a sane default (number of features per tree is normally set
// to sqrt(number of features)).
rf := ensemble.NewRandomForest(10, 2)

// Use cross-fold validation to successively train and evaluate the model
// on 5 folds of the data set.
cv, err :=
evaluation.GenerateCrossFoldValidationConfusionMatrices(irisData, rf, 5)
if err != nil {
    log.Fatal(err)
}
```

运行这个算法得到的准确度比单个决策树更差。如果将每棵树的特征数量改回到4，将重新回归到单个决策树的准确度。这意味着每棵树都用与单个决策树相同的信息进行训练，从而产生相同的结果。

随机森林在这里可能会过度杀伤，并且无法证明任何性能增益的合理性，因此最好坚持使用单一决策树。这个单一的决策树也更易于解释和更有效率。

5.5 朴素贝叶斯

这里将介绍的最后一个分类模型叫做**朴素贝叶斯**（Naive bayes）。在第2章我们讨论了形成此技术基础的贝叶斯规则。朴素贝叶斯是一种基于概率的方法，就像逻辑回归，但其基本思想和假设是不同的。

朴素贝叶斯也在github.com/sjwhitworth/golearn中有实现，可以轻松地尝试它。同样还有其他各种Go的实现，包括github.com/jbrukh/bayesian、github.com/lytics/multibayes和github.com/cdipaolo/goml。

仅供非商业用途或交流学习使用

5.5.1 朴素贝叶斯概念及其重要假设

朴素贝叶斯有一个重要的假设。这个假设表明，类别的概率以及数据集中某个特征的存在与否，与数据集中是否存在其他特征无关。因此，只要提供某些特征存在或不存在，就可以为特定类别的概率写出一个非常简单的公式。

举一个例子来说明这一点。再次假设试图根据电子邮件中的文字预测两类电子邮件，A 和 B（例如是垃圾邮件或者不是垃圾邮件）。朴素贝叶斯会假设某个词的存在 / 不存在与其他词无关。假设某个类包含某些词的概率与所有单独的条件概率相乘成比例。使用这个假设、贝叶斯规则、一些链式规则和独立的假设，可以推导出某个类的条件概率如下：

$$P(B \mid word_1, word_2) \propto P(B)P(word_1 \mid B)P(word_2 \mid B)$$

可以通过计算训练集中特征和标签的出现来计算右侧的所有内容，这是训练模型时所要做的。然后可以通过将这些概率链串起来进行预测。

实际上，在实践中可以使用一个小技巧来避免将一串近似于零的数字串在一起。可以取概率的对数，将它们相加，然后取表达式的指数。这个过程在实践中效果更好。

5.5.2 朴素贝叶斯例子

最后回到贷款数据集，将使用 github.com/sjwhitworth/golearn 中的朴素贝叶斯算法来解决相同的贷款接受问题。将使用在逻辑回归示例中使用的相同的训练和测试集。但是，需要将数据集中的标签转换为 github.com/sjwhitworth/golearn 中使用的二元分类器格式。可以编写一个简单的函数来执行此转换：

```
// convertToBinary utilizes built in golearn functionality to
// convert our labels to a binary label format.
func convertToBinary(src base.FixedDataGrid) base.FixedDataGrid {
    b := filters.NewBinaryConvertFilter()
    attrs := base.NonClassAttributes(src)
    for _, a := range attrs {
        b.AddAttribute(a)
    }
    b.Train()
    ret := base.NewLazilyFilteredInstances(src, b)
    return ret
}
```

一旦有了可以识别的数据，就可以训练和测试朴素贝叶斯模型，如下面的代码所示：

仅供非商业用途或交流学习使用

```
// Read in the loan training data set into golearn "instances".
trainingData, err := base.ParseCSVToInstances("training.csv", true)
if err != nil {
    log.Fatal(err)
}

// Initialize a new Naive Bayes classifier.
nb := naive.NewBernoulliNBClassifier()

// Fit the Naive Bayes classifier.
nb.Fit(convertToBinary(trainingData))

// Read in the loan test data set into golearn "instances".
// This time we will utilize a template of the previous set
// of instances to validate the format of the test set.
testData, err := base.ParseCSVToTemplatedInstances("test.csv", true,
trainingData)
if err != nil {
    log.Fatal(err)
}

// Make our predictions.
predictions := nb.Predict(convertToBinary(testData))

// Generate a Confusion Matrix.
cm, err := evaluation.GetConfusionMatrix(testData, predictions)
if err != nil {
    log.Fatal(err)
}

// Retrieve the accuracy.
accuracy := evaluation.GetAccuracy(cm)
fmt.Printf("\nAccuracy: %0.2f\n\n", accuracy)
```

编译并运行它会得到以下准确度结果：

```
$ go build
$ ./myprogram

Accuracy: 0.63
```

这不像从零开始的逻辑回归那么好。但是，这里仍然有一些预测能力。一个好的练习是从 LendingClub 数据集中添加一些其他特征到这个模型中，特别是一些分类变量。这可能会改善朴素贝叶斯结果。

5.6 参考书目

一般分类：

- `github.com/sjwhitworth/golearn` 文档：https://godoc.org/github.com/sjwhitworth/golearn

仅供非商业用途或交流学习使用

5.7 小结

本章已经涵盖了各种分类模型，包括逻辑回归、k- 最近邻、决策树、随机森林和朴素贝叶斯。事实上，我们甚至从头开始实施了逻辑回归。正如本章所讨论过的，所有这些模型都有其不同的优点和缺点。但是，它们提供了一套优秀的工具集，用于利用 Go 语言执行分类。

在下一章中，将讨论另一种称为**集群**的机器学习。这是将要讨论的第一个无监督技术，并将尝试几种不同的方法。

仅供非商业用途或交流学习使用

第 6 章
集　群

通常，一组数据可以组织成一个集群。例如，可以根据某些潜在属性（例如包括年龄、性别、地理位置、就业状态等的人口统计属性），或者某些潜在流程（例如浏览、购物、机器人交互，以及其他类似的在网站上的行为）组织数据到集群中。用于检测和标记这些集群的机器学习技术自然就被称为**集群**（clustering）技术。

到目前为止，本书讲述的机器学习算法都是**有监督的**（supervised）。也就是说，有一组已知的特征或属性与尝试预测的标签或编号相匹配。利用这些已标记的数据将模型拟合到特定行为上，这些行为是在训练模型之前就已知的。

而大多数集群技术是**无监督的**（unsupervised）。同有监督的回归和分类技术相比，在集群模型找到之前，通常不知道数据集中的集群。因此，下面将利用为未标记的数据集和算法解决集群问题，并使用集群算法为数据自动生成集群标签。

此外与其他机器学习技术相比，很难说给定数据集的*正确*或*准确*集群是什么。不同的集群的数量，以及不同的数据点之间相似度度量的方法，可能会得到各种各样的集群集合，其中每个集群都有一些潜在含义。但这并不意味着集群技术不能用于评估或验证，而是说明需要理解其局限性，并在量化结果时需要小心对待。

6.1　理解集群模型术语

集群非常独特，以下是集群的相关术语。请留意下面的列表仅仅只是一部分，还有许多不同类型的集群技术使用不同的术语。

- **集群**（cluster）或**群组**（group）：每一个集群或群组都是数据点的集合，集群技术是将数据点组织到其中。

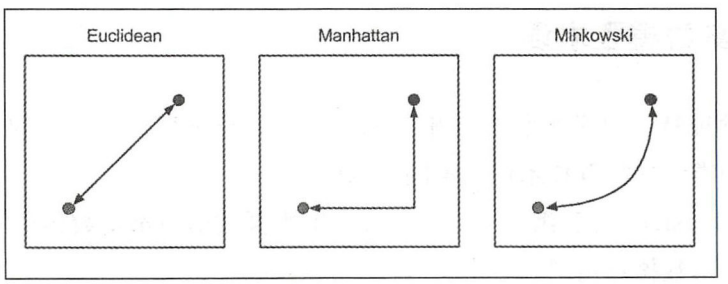
仅供非商业用途或交流学习使用

- **组内**（intra-group）或**集群内**（intra-cluster）：从集群过程中产生的集群，可以通过测量集群内数据点与同集群的数据点之间的相似度来进行评估。这也称为组内或者集群内相似度评估。
- **组间**（inter-group）或**集群间**（inter-cluster）：从集群过程中产生的簇，可以通过测量集群内数据点与不同集群数据点之间的相异度来进行评估。这也称为组间或者集群间相异度评估。
- **内部标准**（internal criteria）：通常没有一套黄金标准的集群标签集可用于评估生成的集群。在这些情况下，可以利用集群内和集群间相似度来衡量集群技术的性能。
- **外部标准**（external criteria）：在其他情况下，有可能有集群标签或分组的黄金标准。例如由人类生成的标准。这些场景中可以使用标准或者外部标准评估集群技术。
- **距离**（distance）或**相似度**（similarity）：这是衡量两个数据点有多接近的度量标准。这可以是特征空间中的欧几里得距离，或者是其他紧密度量度。

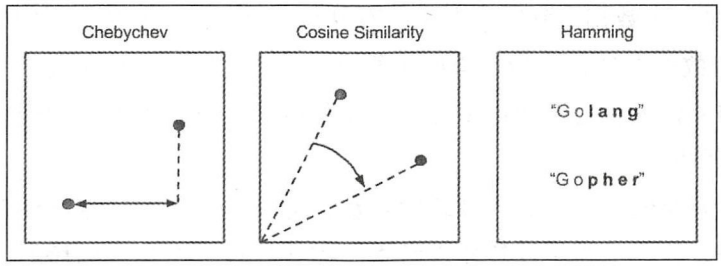
6.2 距离或相似度的度量

为了将数据点集群在一起，需要定义和利用某种距离或相似度，以定量地定义数据点之间的接近程度。这个度量的选择在每个集群项目中都非常重要，因为它直接影响集群的生成方式。使用一种相似度度量产生的集群，可能与另一种相似度度量产生的集群大不相同。

这些距离测量中最常见和简单的方法是**欧几里得距离**（Euclidean distance）或者**平方欧几里得距离**（squared Euclidean distance）。这是特征空间中两个数据点的简单直线距离（你可能还记得这个距离，因为它也用于第 5 章中的 kNN 示例），或者两个点的平方距离。但是仍然有一些其他的，有时会更加复杂的距离度量标准。下图展示了其中一部分。



仅供非商业用途或交流学习使用



例如，**曼哈顿**（Manhattan）距离是两点之间 x 距离的绝对值加上 y 距离的绝对值，**明氏**（Minkowski）距离将**欧几里得**（Euclidean）距离和曼哈顿距离一般化了。与欧几里得距离相比，这些距离度量对于数据中的异常值（或离群值）更加稳健。

其他距离度量（例如**汉明**（Hamming）距离）适合特定类型的数据，例如字符串。在上图的示例中，Golang 和 Gopher 的汉明距离是 4，因为字符串中有 4 个位置的字符不同。因此，如果正在使用文本数据，例如新闻文章或者推文，那么汉明距离可能是一个很好的距离度量标准。

就我们的目标而言，将主要使用欧几里得距离。这个距离可以在 `gonum.org/v1/gonum/floats` 通过 `Distance()` 函数来实现。举个例子，假如要计算位于（1，2）的点同位于（3，4）的点之间的距离，可以如下所示操作。

```
// Calculate the Euclidean distance, specified here via
// the last argument in the Distance function.
distance := floats.Distance([]float64{1, 2}, []float64{3, 4}, 2)

fmt.Printf("\nDistance: %0.2f\n\n", distance)
```

6.3 集群技术的评估

由于不是尝试预测数字或种类，因此之前讨论的连续和离散变量的评估指标并不适用于集群技术。但这并不意味着不去评估集群算法的性能。我们需要知道集群的性能的，这里仅仅介绍一些集群特定的评估指标。

6.3.1 内部集群评估

如果没有用于比较集群的黄金标准集标签，那么可以使用内部标准来评估集群模型的性能。换句话说，仍可以在集群内执行相似度和相异度测量来评估集群。

首先在这里要展示的内部指标称为**轮廓系数**（silhouette coefficient）。如下所示，可以为每个集群的数据点计算轮廓系数。

$$s=\frac{b-a}{\max(a, b)}$$

仅供非商业用途或交流学习使用

这里，a 是一个数据点到同集群中所有其他点之间的平均距离（例如欧几里得距离）。b 是一个数据点到相邻最近集群中所有其他点之间的平均距离。所有数据点轮廓系数的平均值代表在每个集群中点的密集程度。这个平均值可用于每个集群或者所有集群中的数据点。

现在尝试为鸢尾花数据集计算轮廓系数，可以将三个鸢尾花物种看作对应的三个集群的集合。首先，为了计算轮廓数据，需要知道三个集群的**质心**。这些质心只是三个集群（在四维特征空间）中简单的中心点，由此可以确定特定点所在集群的最近集群是什么。

为此，需要解析鸢尾花数据集文件（之前在第 1 章中介绍过），根据集群标签分离数据，平均每个集群的特征，然后计算相应的质心。首先，为质心定义一个 `type` 类型。

```
type centroid []float64
```

然后使用 `github.com/kniren/gota/dataframe` 创建一个包含每个鸢尾花物种质心的集合：

```
// Pull in the CSV file.
irisFile, err := os.Open("iris.csv")
if err != nil {
    log.Fatal(err)
}
defer irisFile.Close()

// Create a dataframe from the CSV file.
irisDF := dataframe.ReadCSV(irisFile)

// Define the names of the three separate species contained in the CSV
file.
speciesNames := []string{
    "Iris-setosa",
    "Iris-versicolor",
    "Iris-virginica",
}

// Create a map to hold our centroid information.
centroids := make(map[string]centroid)

// Filter the dataset into three separate dataframes,
// each corresponding to one of the Iris species.
for _, species := range speciesNames {

    // Filter the original dataset.
    filter := dataframe.F{
        Colname:    "species",
        Comparator: "==",
        Comparando: species,
    }
    filtered := irisDF.Filter(filter)
```

仅供非商业用途或交流学习使用

```
    // Calculate the mean of features.
    summaryDF := filtered.Describe()

    // Put each dimension's mean into the corresponding centroid.
    var c centroid
    for _, feature := range summaryDF.Names() {

        // Skip the irrelevant columns.
        if feature == "column" || feature == "species" {
            continue
        }
        c = append(c, summaryDF.Col(feature).Float()[0])
    }

    // Add this centroid to our map.
    centroids[species] = c
}

// As a sanity check, output our centroids.
for _, species := range speciesNames {
    fmt.Printf("%s centroid: %v\n", species, centroids[species])
}
```

编译并运行以上代码将得到如下质心：

```
$ go build
$ ./myprogram
Iris-setosa centroid: [5.005999999999999 3.4180000000000006 1.464
0.2439999999999999]
Iris-versicolor centroid: [5.936 2.7700000000000005 4.26
1.3259999999999998]
Iris-virginica centroid: [6.587999999999998 2.9739999999999998 5.552 2.026]
```

接下来，需要实际计算每个数据点的轮廓系数。为此修改之前的代码，使得能访问 `for loop` 外层的每个过滤后的数据点集：

```
// Create a map to hold the filtered dataframe for each cluster.
clusters := make(map[string]dataframe.DataFrame)

// Filter the dataset into three separate dataframes,
// each corresponding to one of the Iris species.
for _, species := range speciesNames {

    ...

    // Add the filtered dataframe to the map of clusters.
    clusters[species] = filtered

    ...
}
```

下面创建一个方便的函数从 `dataframe.DataFrame` 的一行中检索浮点数据：

```
// dfFloatRow retrieves a slice of float values from a DataFrame
// at the given index and for the given column names.
func dfFloatRow(df dataframe.DataFrame, names []string, idx int) []float64
{
```

仅供非商业用途或交流学习使用

```
        var row []float64
        for _, name := range names {
                row = append(row, df.Col(name).Float()[idx])
        }
        return row
}
```

下面遍历数据，计算轮廓系数所需的 a 和 b。如下所示，还可以对轮廓系数取平均值，用以得到集群的整体评估指标：

```
// Convert our labels into a slice of strings and create a slice
// of float column names for convenience.
labels := irisDF.Col("species").Records()
floatColumns := []string{
    "sepal_length",
    "sepal_width",
    "petal_length",
    "petal_width",
}

// Loop over the records accumulating the average silhouette coefficient.
var silhouette float64

for idx, label := range labels {

    // a will store our accumulated value for a.
    var a float64

    // Loop over the data points in the same cluster.
    for i := 0; i < clusters[label].Nrow(); i++ {

        // Get the data point for comparison.
        current := dfFloatRow(irisDF, floatColumns, idx)
        other := dfFloatRow(clusters[label], floatColumns, i)

        // Add to a.
        a += floats.Distance(current, other, 2) /
float64(clusters[label].Nrow())
    }
    // Determine the nearest other cluster.
    var otherCluster string
    var distanceToCluster float64
    for _, species := range speciesNames {

        // Skip the cluster containing the data point.
        if species == label {
            continue
        }

        // Calculate the distance to the cluster from the current cluster.
        distanceForThisCluster := floats.Distance(centroids[label],
centroids[species], 2)

        // Replace the current cluster if relevant.
        if distanceToCluster == 0.0 || distanceForThisCluster <
distanceToCluster {
            otherCluster = species
```

```
                distanceToCluster = distanceForThisCluster
            }
        }

        // b will store our accumulated value for b.
        var b float64

        // Loop over the data points in the nearest other cluster.
        for i := 0; i < clusters[otherCluster].Nrow(); i++ {

            // Get the data point for comparison.
            current := dfFloatRow(irisDF, floatColumns, idx)
            other := dfFloatRow(clusters[otherCluster], floatColumns, i)

            // Add to b.
            b += floats.Distance(current, other, 2) /
    float64(clusters[otherCluster].Nrow())
        }

        // Add to the average silhouette coefficient.
        if a > b {
            silhouette += ((b - a) / a) / float64(len(labels))
        }
        silhouette += ((b - a) / b) / float64(len(labels))
    }

    // Output the final average silhouette coeffcient to stdout.
    fmt.Printf("\nAverage Silhouette Coefficient: %0.2f\n\n", silhouette)
```

编译和运行此示例将得到以下结果：

```
$ go build
$ ./myprogram

Average Silhouette Coefficient: 0.51
```

如何得知平均轮廓系数 0.51 是好是坏？请记住轮廓系数与集群内平均距离和集群间平均距离的差值成正比，它总位于 0.0 和 1.0 之间。因此，更高的值（接近 1.0 的值）意味着更密集的集群，并且与其他集群更加不同。

在通常情况下，需要调整集群的数量或者集群的技术用以优化轮廓分数（使其更大）。在这里实际处理的是已经手工标记过的数据，因此 0.51 对这个数据集是个很好的分数。在其他数据集中，可能更高或更低，这取决于数据中集群是否存在，以及相似性度量指标的选择。

轮廓系数绝不是内部评估集群的唯一方法。可以利用轮廓分数中的 a 和 b 的数量来评估本集群或者其他集群的同质性，以及不同集群之间的差异性。同样还可以使用集群内点到集群质心的平均距离去测量集群的密度。更近一步，还可以使用各种其他评估指标，这些指标在此不做详述，例如 Calinski-Harabaz index（在 http://datamining.rutgers.edu/publication/internalmeasures.pdf 中有详细的讨论）。

仅供非商业用途或交流学习使用

6.3.2 外部集群评估

如果对集群有一个基本事实或黄金标准，那么可以利用各种外部集群评估技术。这个基本事实或者黄金标准意味着可以访问或可以得到（通过人工注释）一组标示为真或需要的集群标签的数据。

通常并没有这样关于集群的黄金标准，因此这里不会详细介绍这些评估技术。但是，如果读者感兴趣，可以查看 Adjusted Rand index、Mutual Information、Fowlkes-Mallows scores、completeness 以及 V-measures。这些是相关的外部集群评估指标（更多信息请查询 https://nlp.stanford.edu/IR-book/html/htmledition/evaluation-of-clustering-1.html）。

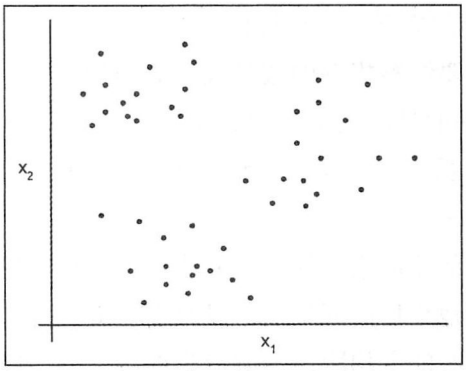
6.4 k-均值集群

这里第一个要介绍的集群技术也许是最知名的集群技术，称为 **k-均值集群**（k-means clustering）或者 **k-均值**（k-means）。k-均值是一种迭代方法，其中数据点集群在每次迭代的集群质心周围。这项技术相对容易掌握，但是有一些相关细节还是容易被忽视，因此在探索该技术时会着重强调。

由于 k-均值集群非常容易实现，因此 Go 中有许多概念验证的算法实现。可以在这个链接中搜索到 k-均值集群（https://golanglibs.com/top?q=kmeans）。但是，这里将使用最新并且相对简单的实现，即 github.com/mash/gokmeans。

6.4.1 k-均值集群综述

假设已知由两个变量 x_1 和 x_2 定义的一组数据点。这些数据点自然地能显示一些分组到集群中，如下图所示。

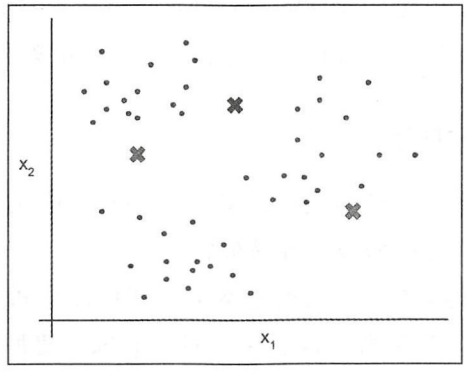


为了使用 k-均值自动集群这些点，首先要知道需从集群操作中获得多少集群。这便是参数 k，也是 k-均值名称的来历。在本例中，使用 $k=3$。

然后随机选择 k 个质心的 x_1 和 x_2 位置，这些随机质心将作为算法的起点。下图

仅供非商业用途或交流学习使用

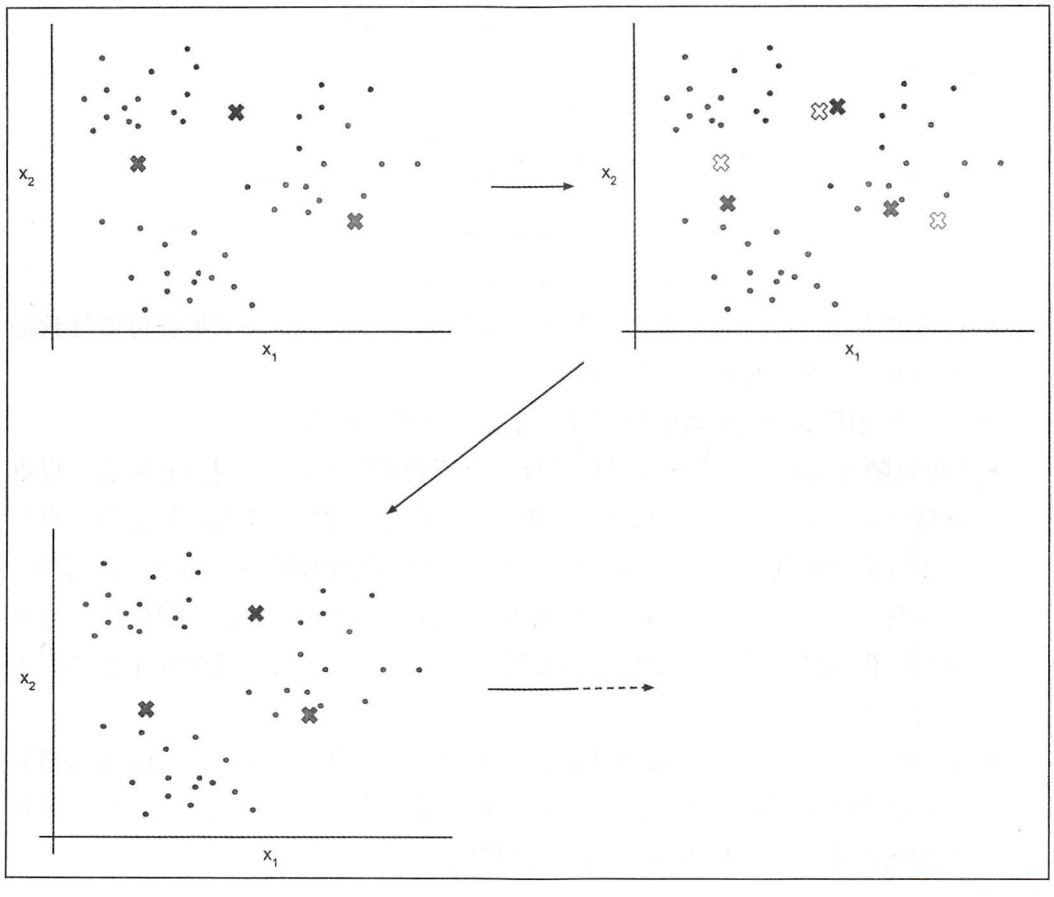
通过 X 展示了这种随机质心。



为了优化这些质心并且集群数据点，需要迭代执行以下操作：

1. 将每个数据点分配给最近质心对应的集群（这里测量的是欧几里得距离）。
2. 在每个集群中计算 x_1 和 x_2 的位置。
3. 根据计算出的 x_1 和 x_2 位置更新每个质心位置。

重复步骤 1～3，直到步骤 1 中的分配不再更改，该过程如下所示。



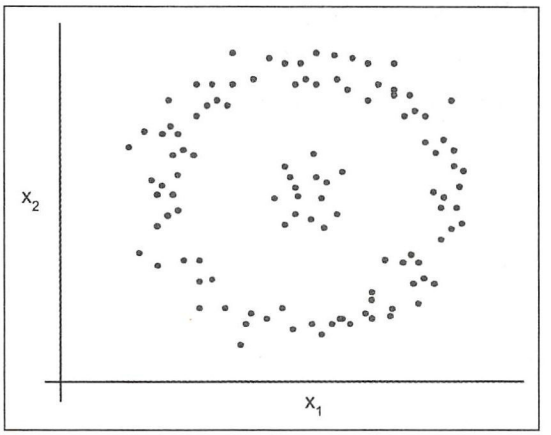
仅供非商业用途或交流学习使用

这里仅仅用一个静态图加以说明，希望能对大家有所帮助。但是如果想可视化 k-均值过程的每个步骤，以更好地理解质心变化，可以查看 http://stanford.edu/class/ee103/visualizations/kmeans/kmeans.Html，其中包含了 k-均值集群过程的可交互的动画。

6.4.2 k-均值的假设和陷阱

k-均值看起来可能像一个非常简单的算法，事实也确实如此。然而，它需要对数据做出一些基本假设，这些假设很容易被忽略。

- **球形或空间分组的集群**：k-均值基本上是在特征空间中绘制球形或空间上接近的区域，用以找到集群。这意味着对于非球形集群（也就是说，集群看起来不像特征空间中的分组区块），k-均值可能会失败。具体来说，k-均值可能表现不佳的非球形集群如下所示：



- **相似的大小**：k-均值也假定集群都是相似的大小。小的外围集群可能导致简单的 k-均值算法偏离过程从而产生奇怪的分组。

此外，当使用 k-均值来集群数据时，可能会遇到一些陷阱：

- k 可以随意选择。这意味着可以选择一个不合逻辑的 k，同时也意味着可以继续增加 k，直到对每个点都有一个集群（这将是非常好的集群，因为每个点与它本身完全相同）。为了指导 k 的选择，应该使用**肘形图**（elbow graph）方法。在这种方法中，计算评估指标时增加 k。随着 k 的增加，评估指标应该越来越好，但最终会出现一个拐点，表明收益递减。理想的 k 就在这个拐点，也称为肘部。
- 不能保证 k-均值总是会收敛到相同的集群。当从随机质心开始时，k 均值算法可能在不同的运行中收敛到不同的局部最小值。应该意识到这一点，并从各种初始化中运行 k-均值算法以确保稳定性。

仅供非商业用途或交流学习使用

6.4.3 k-均值集群的例子

这里用送货司机的数据集说明集群技术，数据集如下所示：

```
$ head fleet_data.csv
Driver_ID,Distance_Feature,Speeding_Feature
3423311935,71.24,28.0
3423313212,52.53,25.0
3423313724,64.54,27.0
3423311373,55.69,22.0
3423310999,54.58,25.0
3423313857,41.91,10.0
3423312432,58.64,20.0
3423311434,52.02,8.0
3423311328,31.25,34.0
```

第一列 Driver_ID 包括特定司机的各种匿名标识。第二列和第三列是将在集群中使用的属性。Distance_Feature 列是每个数据的平均驾驶距离，而 Speeding_Feature 是司机时速超过限速 5 英里（1 英里 =1609.344 米）以上的平均时间百分比。

集群的目标是根据 Distance_Feature 和 Speeding_Feature 将送货司机分组。请记住，这是一种无监督的学习技术，因此没有人知道数据中应该或可能形成哪些集群。希望在练习开始时了解一些关于司机的知识。

6.4.3.1 分析数据

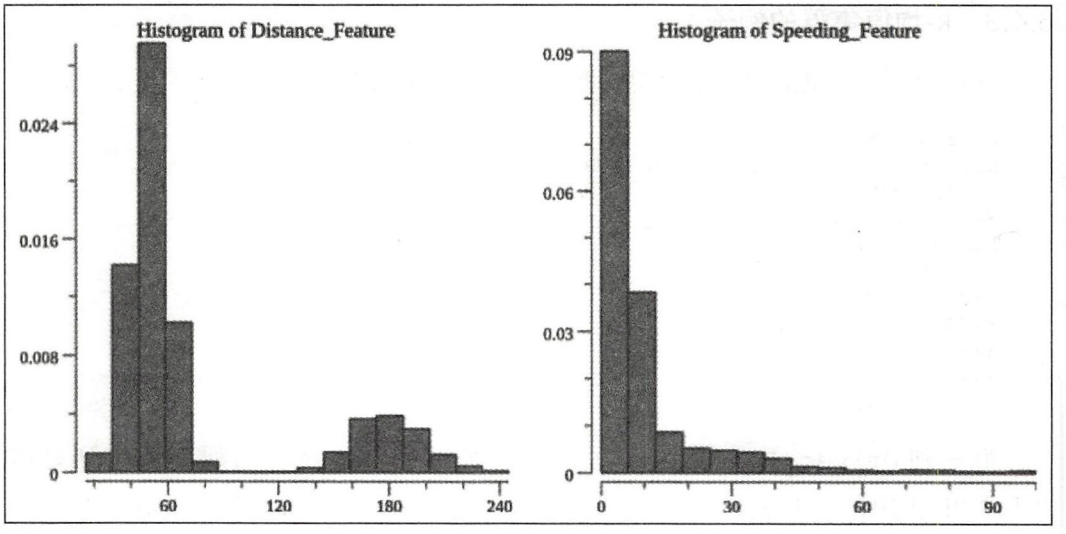
是的，你猜对了！这里有一个新的数据集，需要对这个数据集进行剖析以更多地了解它。首先使用 github.com/kniren/dataframe 计算汇总统计信息，并使用 gonum.org/v1/plot 创建每个特征的直方图。我们已经在第 4 章和第 5 章中多次完成了这一操作，所以在此不会重新编写代码。来一起看看结果：

```
$ go build
$ ./myprogram
[7x4] DataFrame

    column   Driver_ID          Distance_Feature Speeding_Feature
 0: mean     3423312447.500000 76.041523        10.721000
 1: stddev   1154.844867       53.469563        13.708543
 2: min      3423310448.000000 15.520000        0.000000
 3: 25%      3423311447.000000 45.240000        4.000000
 4: 50%      3423312447.000000 53.330000        6.000000
 5: 75%      3423313447.000000 65.610000        9.000000
 6: max      3423314447.000000 244.790000       100.000000
    <string> <float>           <float>          <float>
```

哇！看起来大多数司机的超速时间是 10%，这有点不可思议。一位司机甚至似乎 100% 的时间都在超速。希望我不在他行驶的路线上。

特征的直方图如下所示：

仅供非商业用途或交流学习使用



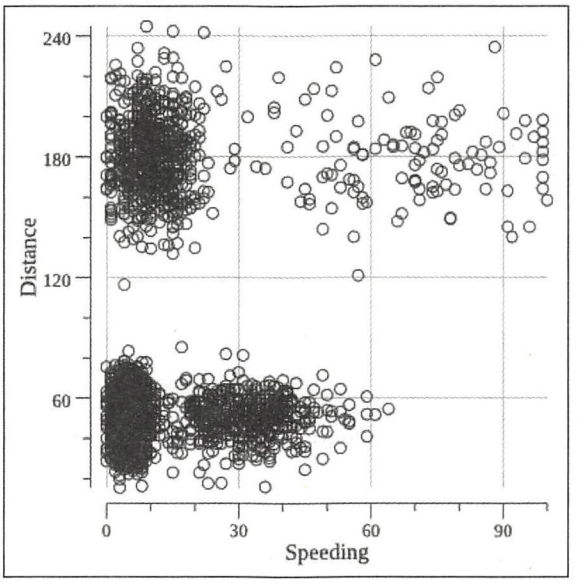
看起来 Distance_Feature 数据中好像有一个有趣的结构。实际上这将很快融入集群中，但是可以通过创建特征空间的散点图来获得这种结构的另一个视图：

```
// Open the driver dataset file.
f, err := os.Open("fleet_data.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a dataframe from the CSV file.
driverDF := dataframe.ReadCSV(f)

// Extract the distance column.
yVals := driverDF.Col("Distance_Feature").Float()

// pts will hold the values for plotting
pts := make(plotter.XYs, driverDF.Nrow())

// Fill pts with data.
for i, floatVal := range driverDF.Col("Speeding_Feature").Float() {
    pts[i].X = floatVal
    pts[i].Y = yVals[i]
}

// Create the plot.
p, err := plot.New()
if err != nil {
    log.Fatal(err)
}
p.X.Label.Text = "Speeding"
p.Y.Label.Text = "Distance"
p.Add(plotter.NewGrid())

s, err := plotter.NewScatter(pts)
if err != nil {
    log.Fatal(err)
```

仅供非商业用途或交流学习使用


```
}
s.GlyphStyle.Color = color.RGBA{R: 255, B: 128, A: 255}
s.GlyphStyle.Radius = vg.Points(3)

// Save the plot to a PNG file.
p.Add(s)
if err := p.Save(4*vg.Inch, 4*vg.Inch, "fleet_data_scatter.png"); err !=
nil {
    log.Fatal(err)
}
```

编译并运行它将创建以下散点图：



相比直方图，这看起来会更加结构化。看起来这里至少有两组清晰的数据集群。这种关于数据的直觉可以在正式应用集群技术时作为直观的检查，它可以提供一个 k 值的尝试起点。

6.4.3.2 用 k-均值生成集群

现在通过实际应用 k-均值集群到送货司机数据来实战一下。为了利用 `github.com/mash/gokmeans`，首先需要创建一个 `gokmeans.Node` 值的数据切片，这些值将被输入到集群中：

```
// Open the driver dataset file.
f, err := os.Open("fleet_data.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a new CSV reader.
r := csv.NewReader(f)
r.FieldsPerRecord = 3
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
// Initialize a slice of gokmeans.Node's to
// hold our input data.
var data []gokmeans.Node

// Loop over the records creating our slice of
// gokmeans.Node's.
for {

    // Read in our record and check for errors.
    record, err := r.Read()
    if err == io.EOF {
        break
    }
    if err != nil {
        log.Fatal(err)
    }

    // Skip the header.
    if record[0] == "Driver_ID" {
        continue
    }

    // Initialize a point.
    var point []float64

    // Fill in our point.
    for i := 1; i < 3; i++ {

        // Parse the float value.
        val, err := strconv.ParseFloat(record[i], 64)
        if err != nil {
            log.Fatal(err)
        }

        // Append this value to our point.
        point = append(point, val)
    }

    // Append our point to the data.
    data = append(data, gokmeans.Node{point[0], point[1]})
}
```

然后，生成集群就像调用 gomeans.Train(...) 函数一样简单。具体来说，将用 k=2 和最多 50 次迭代来调用这个函数：

```
// Generate our clusters with k-means.
success, centroids := gokmeans.Train(data, 2, 50)
if !success {
    log.Fatal("Could not generate clusters")
}

// Output the centroids to stdout.
fmt.Println("The centroids for our clusters are:")
for _, centroid := range centroids {
    fmt.Println(centroid)
}
```

运行这些，将得到生成的集群的质心：

仅供非商业用途或交流学习使用

```
$ go build
$ ./myprogram
The centroids for our clusters are:
[50.04763437499999 8.82875]
[180.01707499999992 18.29]
```

太好了！已经生成了第一批集群。现在，需要继续评估这些集群的合理性。

刚刚输出了集群的质心，这正是数据点分组所需的所有信息。如果想知道数据点是在第一个还是第二个集群中，只需要计算离质心的距离。这个点属于最接近质心对应的组。

6.4.3.3 评估生成的集群

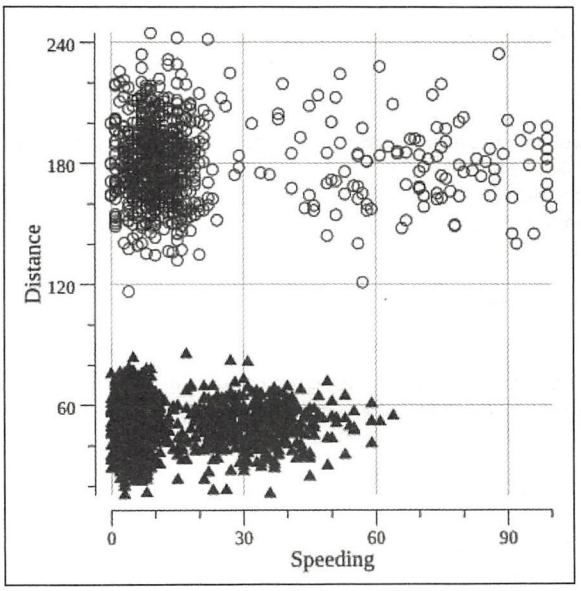
第一种方法可以直观地评估刚刚生成的集群。创建另一个散点图，然而这次为每个组使用不同的形状：

```
// Open the driver dataset file.
f, err := os.Open("fleet_data.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a dataframe from the CSV file.
driverDF := dataframe.ReadCSV(f)

// Extract the distance column.
yVals := driverDF.Col("Distance_Feature").Float()

// clusterOne and clusterTwo will hold the values for plotting.
var clusterOne [][]float64
var clusterTwo [][]float64

// Fill the clusters with data.
for i, xVal := range driverDF.Col("Speeding_Feature").Float() {
    distanceOne := floats.Distance([]float64{yVals[i], xVal},
[]float64{50.05, 8.83}, 2)
    distanceTwo := floats.Distance([]float64{yVals[i], xVal},
[]float64{180.02, 18.29}, 2)
    if distanceOne < distanceTwo {
        clusterOne = append(clusterOne, []float64{xVal, yVals[i]})
        continue
    }
    clusterTwo = append(clusterTwo, []float64{xVal, yVals[i]})
}

// pts* will hold the values for plotting
ptsOne := make(plotter.XYs, len(clusterOne))
ptsTwo := make(plotter.XYs, len(clusterTwo))

// Fill pts with data.
for i, point := range clusterOne {
    ptsOne[i].X = point[0]
    ptsOne[i].Y = point[1]
```

仅供非商业用途或交流学习使用

```
}

for i, point := range clusterTwo {
    ptsTwo[i].X = point[0]
    ptsTwo[i].Y = point[1]
}

// Create the plot.
p, err := plot.New()
if err != nil {
    log.Fatal(err)
}
p.X.Label.Text = "Speeding"
p.Y.Label.Text = "Distance"
p.Add(plotter.NewGrid())

sOne, err := plotter.NewScatter(ptsOne)
if err != nil {
    log.Fatal(err)
}
sOne.GlyphStyle.Radius = vg.Points(3)
sOne.GlyphStyle.Shape = draw.PyramidGlyph{}

sTwo, err := plotter.NewScatter(ptsTwo)
if err != nil {
    log.Fatal(err)
}
sTwo.GlyphStyle.Radius = vg.Points(3)
// Save the plot to a PNG file.
p.Add(sOne, sTwo)
if err := p.Save(4*vg.Inch, 4*vg.Inch, "fleet_data_clusters.png"); err !=
nil {
    log.Fatal(err)
}
```

此代码生成以下散点图，这清楚地显示了成功做好的集群分类：



仅供非商业用途或交流学习使用

定性地说，可以看到一个司机集群的驾驶距离较短，而另一个司机集群的驾驶距离较长。这些实际上分别对应于农村和城市送货司机（或短途和长途司机）。

为了更加定量地评估集群，可以计算集群中的点与集群质心之间的集群内平均距离。为了完成这项工作，这里会创建一个便利的函数：

```
// withinClusterMean calculates the mean distance between
// points in a cluster and the centroid of the cluster.
func withinClusterMean(cluster [][]float64, centroid []float64) float64 {

    // meanDistance will hold our result.
    var meanDistance float64

    // Loop over the points in the cluster.
    for _, point := range cluster {
            meanDistance += floats.Distance(point, centroid, 2) /
    float64(len(cluster))
        }

        return meanDistance
    }
```

现在，为了评估集群，只需要为每个集群调用这个函数：

```
// Output our within cluster metrics.
fmt.Printf("\nCluster 1 Metric: %0.2f\n", withinClusterMean(clusterOne,
[]float64{50.05, 8.83}))
fmt.Printf("\nCluster 2 Metric: %0.2f\n", withinClusterMean(clusterTwo,
[]float64{180.02, 18.29}))
```

运行该代码会得到以下指标：

```
$ go build
$ ./myprogram

Cluster 1 Metric: 11.68

Cluster 2 Metric: 23.52
```

正如所看到的，集群 1（散点图中的浅红色集群）大约是集群 2 的两倍（紧凑）。在没有图表的情况下也是如此，并给了我们更多关于集群的定量信息。

请注意，显然此处我们正在寻找两个集群。但是，在其他情况下，集群的数量可能并不明确，特别是在许多特征无法可视化的情况下。在这些时候，使用一种方法（如肘形法）来确定正确的 k 就尤其重要。有关此方法的更多信息，请访问 https://datasciencelab.wordpress.com/2013/12/27/finding-the-k-in-k-means-clustering/。

6.5 其他集群技术

还有其他一些集群技术，这里不做更多讨论。其中包括 DBSCAN 和分层集群。

仅供非商业用途或交流学习使用

不幸的是，当前 Go 的实现仅限于这些集群技术。DBSCAN 在 https://github.com/sjwhitworth/golearn 中实现。但是据我所知，目前还没有其他集群技术的实现。

这给为社区做出贡献创造了很好的机会！集群技术通常并不复杂，创建另一种集群技术的实现是回馈 Go 数据科学社区的好方法。如果想讨论实施、提出问题或寻求帮助，请随时联系 Gophers Slack（`@dwhitena`）中的作者或 Gophers Slack 上 `#data-science` 中的其他数据科学专家。

6.6 参考书目

距离度量和评估集群：

- 集群评估概述：https://nlp.stanford.edu/IR-book/html/htmledition/evaluation-of-clustering-1.html
- 各种距离 / 相似性度量的比较：http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0144059
- 可视化 k- 均值集群：https://www.naftaliharris.com/blog/visualizing-k-means-clustering/
- `github.com/mash/gokmeans` 文档：https://godoc.org/github.com/mash/gokmeans

6.7 小结

在本章中，我们介绍了集群的一般原理，学会了如何评估生成的集群，并学习了如何使用 Go 语言实现 k-均值集群。大家现在应该明白怎么检测数据集中的分组结构了。

接下来，我们将讨论时间序列数据的建模，比如股票价格、传感器数据，等等。

仅供非商业用途或交流学习使用

第 7 章

时间序列和异常检测

之前讨论的大部分模型是基于相关属性预测某种特性。例如，基于花朵的测量结果预测其种类，还尝试根据患者的医疗特征预测其糖尿病的进展情况。

时间序列建模的前提不同于这些属性预测模型。简而言之，时间序列建模有助于我们根据过去的属性来预测未来。例如，可以根据股票的历史价格预测未来价格。或者想要根据网站的历史用户量来预测网站在特定时间的用户量。以上这些有时也称为**预报**（forecasting）。

时间序列建模中使用的数据通常与分类、回归或集群中使用的数据不同。正如人们所期望的那样，时序模型运行在一个或多个**时间序列**上。该序列是一系列的内容、属性或者其他数字，它们与相应的日期和时间配对，抑或与日期和时间的相应代理（例如测量索引或者日期编号）配对。就股票价格而言，这个序列会包含一批（日期和时间、股票价格）配对。

这个时间序列数据在工业界和学术界无处不在。随着对**物联网**（IoT）的探索和开发，它变得越来越重要。健身追踪、冰箱、恒温器、照相机、无人机等新出现的智能设备，正产生着数量惊人的时序数据。

当然，对这种类型的数据并不局限于预测未来。还可以利用时序数据做许多有用的事情，其中包括本章稍后会谈到的异常检测。异常检测尝试检测一个时间序列中的意外或非正常事件。这些事件可能是一些同灾难性天气、基础设施失效、病毒式社交行为等有关的事情。

7.1 在 Go 中表示时序数据

有专门的系统用来存储和处理时序数据。其中一些是用 Go 编写的，包括

仅供非商业用途或交流学习使用

Prometheus 和 InfluxDB。另外还有一些本书已经介绍过的工具同样能处理时间序列。具体来说，github.com/kniren/gota/dataframe、gonum.org/v1/gonum/floats 及 gonum.org/v1/gonum/mat 有助于处理时序数据。

以一个包含时间序列的数据集为例，该序列表示 1949—1960 年间国际航空旅客数量（可从此处获得：https://raw.github.com/vincentarelbundock/Rdatasets/master/csv/datasets/Air-Passengers.csv）：

```
$ head AirPassengers.csv
time,AirPassengers
1949.0,112
1949.08333333,118
1949.16666667,132
1949.25,129
1949.33333333,121
1949.41666667,135
1949.5,148
1949.58333333,148
1949.66666667,136
```

这里，time 列包含一个用带小数的年份所表示的时间序列。AirPassengers 列包含 time 所示时间点的国际乘客数量。换句话说，这是一个（时间，乘客数量）配对的时间序列。

这只是表格数据，用数据框或者矩阵来表示更好。为简单起见，这里使用数据框，如下面的代码所示。

```
// Open the CSV file.
passengersFile, err := os.Open("AirPassengers.csv")
if err != nil {
    log.Fatal(err)
}
defer passengersFile.Close()

// Create a dataframe from the CSV file.
passengersDF := dataframe.ReadCSV(passengersFile)

// As a sanity check, display the records to stdout.
// Gota will format the dataframe for pretty printing.
fmt.Println(passengersDF)
```

这会得到如下输出：

```
$ go build
$ ./myprogram
[144x2] DataFrame

    time        AirPassengers
 0: 1949.000000 112
 1: 1949.083333 118
 2: 1949.166667 132
 3: 1949.250000 129
 4: 1949.333333 121
 5: 1949.416667 135
```

仅供非商业用途或交流学习使用

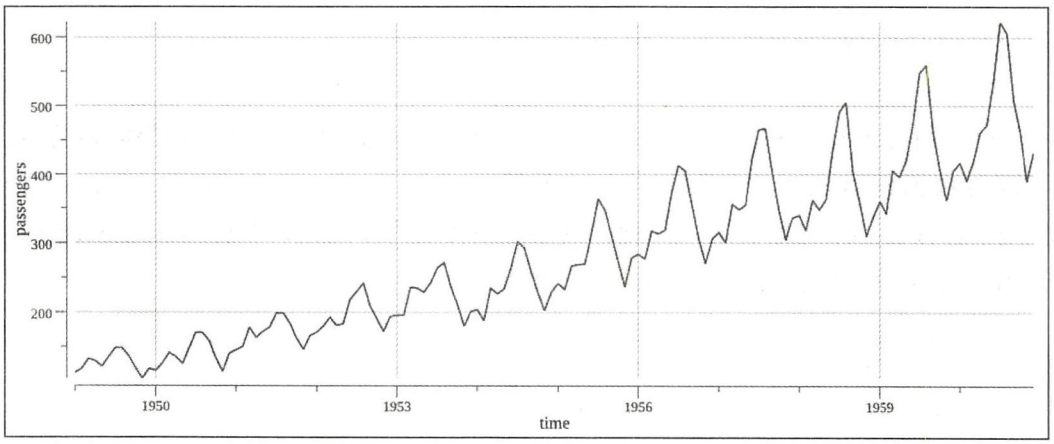
```
 6: 1949.500000 148
 7: 1949.583333 148
 8: 1949.666667 136
 9: 1949.750000 119
    ...          ...
    <float>      <int>
```

可以用 gonum.org/v1/gonum/mat 近似地表示这个序列。当 / 如果需要的话，可以将数据框转换为浮点的切片，以便同 gonum.org/v1/gonum/floats 结合使用。例如，如果想绘制时间序列，可以将列转换为浮点数，然后用 gonum.org/v1/plot 生成一个绘图。如下面的代码所示：

```
// Open the CSV file.
passengersFile, err := os.Open("AirPassengers.csv")
if err != nil {
    log.Fatal(err)
}
defer passengersFile.Close()

// Create a dataframe from the CSV file.
passengersDF := dataframe.ReadCSV(passengersFile)

// Extract the number of passengers column.
yVals := passengersDF.Col("AirPassengers").Float()

// pts will hold the values for plotting.
pts := make(plotter.XYs, passengersDF.Nrow())

// Fill pts with data.
for i, floatVal := range passengersDF.Col("time").Float() {
    pts[i].X = floatVal
    pts[i].Y = yVals[i]
}

// Create the plot.
p, err := plot.New()
if err != nil {
    log.Fatal(err)
}
p.X.Label.Text = "time"
p.Y.Label.Text = "passengers"
p.Add(plotter.NewGrid())

// Add the line plot points for the time series.
l, err := plotter.NewLine(pts)
if err != nil {
    log.Fatal(err)
}
l.LineStyle.Width = vg.Points(1)
l.LineStyle.Color = color.RGBA{B: 255, A: 255}

// Save the plot to a PNG file.
p.Add(l)
if err := p.Save(10*vg.Inch, 4*vg.Inch, "passengers_ts.png"); err != nil {
    log.Fatal(err)
}
```

仅供非商业用途或交流学习使用

编译并运行这个程序可以产生如下时序图：



正如预期的那样，由于越来越多的人通过飞机旅行，国际航空旅客的数量随着时间的推移而增加。还可以看到随着时间推移，有明显的颠簸或波峰出现。我们很快会深入探索这些情况。

7.2 理解时间序列的术语

读者可能注意到了，在本书中每一套机器学习技术都有一套相关的术语，时间序列也是如此。

以下是本章其余部分将要使用的一些术语的解释。

- **时间、日期时间或时间戳**：该属性是时间序列中每个匹配对的临时元素。这可能是一个简单的时间，或者是日期和时间的组合（有时称为日期时间或者时间戳），它还可能包括时区。
- **观察、测量、信号或随机变量**：这是我们尝试预测或作为时间函数分析的属性。
- **季节性**：例如航空旅客数据的时间序列会随着季节（周、月、年等）不同而发生变化。时间序列的这种行为被认为具有季节性。
- **趋势**：随时间逐渐增加或减少的时间序列（与季节效应分开）所呈现的一种趋势。
- **平稳的**：随着时间推移，时间序列呈现出相同的模式，没有趋势或者其他渐变（例如方差或者协方差的变化），这种情况被称为平稳的。
- **时间段**：时间序列中连续观察点之间的时间量，或者序列中一个时间戳与先前发生的时间戳之间的差异。

仅供非商业用途或交流学习使用

- **自回归模型**：这是一种模型，试图通过同一过程的一个或多个延迟或滞后版本对时间序列过程建模。例如股票价格的自回归模型会尝试用前一时间点的股票价格对股票价格建模。
- **移动平均模型**：这是一个模型，它试图根据不完全预测项（通常称为**误差**）的当前和过去值对时间序列进行建模。例如，这个不完全预测项可能是时间序列中的一些白噪。

7.3　与时间序列有关的统计

除了与时间序列相关的某些术语外，还有一系列与时间序列有关的重要统计数据。我们将在执行预测和异常检测时依靠这些统计数据。这些统计数据主要同时间序列中的值如何与同一序列中的其他值相关联有关。

这些统计数据将有助于分析数据，这是任何时间序列建模项目的重要组成部分，因为它与我们所涵盖的所有其他类型的建模类似。对时间序列的行为与时间、季节性和趋势的关系的直觉，可以确保应用适当的模型和对结果进行心理检查。

7.3.1　自相关

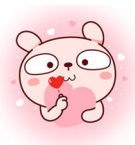
自相关是衡量信号与其本身的延迟版本之间相关性的度量。例如，一次或多次对股票价格的观察可能与股票价格的下一次观察相关联（或一起改变）。如果是这样的话，我们就说股价受到一定的滞后或延迟的影响。然后，可以通过股价的延迟版本对未来股价进行建模，特定的滞后时间是与股价紧密相关的。

为了测量变量 x_t 与其自身延迟版本（或滞后版本）x_s 的自相关性，可以利用**自相关函数**（ACF），定义如下：

$$acf=\frac{(x_{t1}-\bar{x}_t)(x_{s1}-\bar{x}_t)+(x_{t2}-\bar{x}_t)(x_{s1}-\bar{x}_t)+\cdots}{(x_{t1}-\bar{x}_t)^2+(x_{t2}-\bar{x}_t)^2+\cdots}\begin{cases}x_t \text{ 表示序列}\\ x_s \text{ 表示一个延时或滞后的时序}\\ \bar{x}_t \text{ 表示 } x_t \text{ 的平均值}\end{cases}$$

在这里，s 可以表示任何版本的 x。因此可以计算 x 与其一个延迟时间段版本（x_{t-1}）或者 x 与其两个延迟时间段版本（x_{t-2}）等的自相关性。这样可以得知 x 与哪个延迟版本最相关，从而确定 x 的哪些延迟版本可能是用于建模未来版本 x 的良好候选者。

下面来尝试计算航空公司乘客时间序列的前几个自相关。为了做到这一点，首先需要创建一个函数来计算时间序列中特定时间延迟的自相关。如下是这个函数的实现示例：



仅供非商业用途或交流学习使用


```
// acf calculates the autocorrelation for a series
// at the given lag.
func acf(x []float64, lag int) float64 {

    // Shift the series.
    xAdj := x[lag:len(x)]
    xLag := x[0 : len(x)-lag]

    // numerator will hold our accumulated numerator, and
    // denominator will hold our accumulated denominator.
    var numerator float64
    var denominator float64

    // Calculate the mean of our x values, which will be used
    // in each term of the autocorrelation.
    xBar := stat.Mean(x, nil)

    // Calculate the numerator.
    for idx, xVal := range xAdj {
        numerator += ((xVal - xBar) * (xLag[idx] - xBar))
    }

    // Calculate the denominator.
    for _, xVal := range x {
        denominator += math.Pow(xVal-xBar, 2)
    }

    return numerator / denominator
}
```

然后遍历几个延迟，利用 `acf()` 函数来计算各种自相关。如下代码显示了此过程：

```
// Open the CSV file.
passengersFile, err := os.Open("AirPassengers.csv")
if err != nil {
    log.Fatal(err)
}
defer passengersFile.Close()

// Create a dataframe from the CSV file.
passengersDF := dataframe.ReadCSV(passengersFile)

// Get the time and passengers as a slice of floats.
passengers := passengersDF.Col("AirPassengers").Float()

// Loop over various values of lag in the series.
fmt.Println("Autocorrelation:")
for i := 1; i < 11; i++ {

    // Shift the series.
    adjusted := passengers[i:len(passengers)]
    lag := passengers[0 : len(passengers)-i]

    // Calculate the autocorrelation.
    ac := stat.Correlation(adjusted, lag, nil)
    fmt.Printf("Lag %d period: %0.2f\n", i, ac)
}
```

仅供非商业用途或交流学习使用

该过程会得到如下结果：

```
$ go build
$ ./myprogram
Autocorrelation:
Lag 1 period: 0.95
Lag 2 period: 0.88
Lag 3 period: 0.81
Lag 4 period: 0.75
Lag 5 period: 0.71
Lag 6 period: 0.68
Lag 7 period: 0.66
Lag 8 period: 0.66
Lag 9 period: 0.67
Lag 10 period: 0.70
```

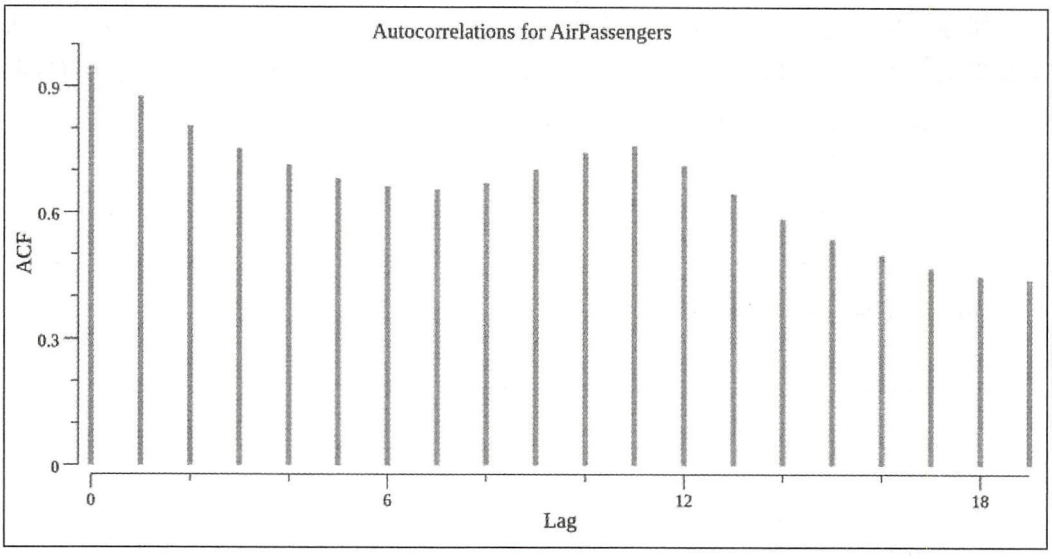
如你所见，这个序列中随着延迟增大，其自相关趋势越来越小（尽管并不是每个延迟都是如此）。然而，这些信息在其数字形式中可能有点难以吸收。我们将这些数值绘制为延迟函数，以更好地显示相关性。

```
// Open the CSV file.
passengersFile, err := os.Open("AirPassengers.csv")
if err != nil {
    log.Fatal(err)
}
defer passengersFile.Close()

// Create a dataframe from the CSV file.
passengersDF := dataframe.ReadCSV(passengersFile)

// Get the time and passengers as a slice of floats.
passengers := passengersDF.Col("AirPassengers").Float()

// Create a new plot, to plot our autocorrelations.
p, err := plot.New()
if err != nil {
    log.Fatal(err)
}

p.Title.Text = "Autocorrelations for AirPassengers"
p.X.Label.Text = "Lag"
p.Y.Label.Text = "ACF"
p.Y.Min = 0
p.Y.Max = 1

w := vg.Points(3)

// Create the points for plotting.
numLags := 20
pts := make(plotter.Values, numLags)

// Loop over various values of lag in the series.
for i := 1; i <= numLags; i++ {

    // Calculate the autocorrelation.
    pts[i-1] = acf(passengers, i)
}
```

仅供非商业用途或交流学习使用


```
// Add the points to the plot.
bars, err := plotter.NewBarChart(pts, w)
if err != nil {
    log.Fatal(err)
}
bars.LineStyle.Width = vg.Length(0)
bars.Color = plotutil.Color(1)

// Save the plot to a PNG file.
p.Add(bars)
if err := p.Save(8*vg.Inch, 4*vg.Inch, "acf.png"); err != nil {
    log.Fatal(err)
}
```

这些代码将产生如下 ACF 图：



请注意，整体来看自相关是呈下降趋势，但它们的值一直很大（远高于 0.5），甚至在 20 个时间段之后仍然很高。这表明我们的时间序列不平稳。事实上，如果看一下之前的时间序列图，它显然是呈上升趋势。我们将在本章后面讨论这种不平稳的行为，但现在，足以说明 ACF 图表向我们展示：延迟版本的航空旅客数量与未延迟的旅客数量相关。

更一般而言，ACF 使得我们可以确定正在为哪种类型的时间序列建模。对于一个可以通过自回归模型很好建模的过程，可以看到随着延迟进一步扩大，acf 函数会稍微快速下降，但不会是立刻下降。对于一个可以通过所谓移动平均模型很好地建模的过程，会在第一次延迟之后看到一个重要的 ACF 项，但是 ACF 将在第一次延迟后消失。

关于 ACF 图表的更多信息，请参考 https://coolstatsblog.com/2013/08/07/how-to-

仅供非商业用途或交流学习使用

use-the-autocorreation-function-acf/，其中提供了一些很好的细节介绍，包括本书中不会涵盖的部分。

7.3.2　偏自相关

正如能从名字中预测的那样，偏自相关与自相关有关，但是有一些细微差别。偏意味着这是一种有条件的关联。从本质上讲，偏自相关是测量一个序列在减去任何中间延迟的自相关后，在特定延迟步长的相关性。可以将其视为移除中间关联后的剩余自相关。

这么做的原因是，我们期望的不仅仅是使用 ACF 来确定时序模型的顺序，而且还需要它能够通过自回归模型建模。假设使用 ACF，因为 ACF 会随着延迟呈指数衰减，所以已经确定可以使用自回归模型为序列建模。那么如何得知，是使用一个时间段的延迟版本还是同时使用一个时间段和两个时间段的延迟版本等为该时间序列进行建模？

通过减去中间相关性，可以快速确定任何剩余相关性是否可以使用具有更多项的自回归模型进行建模。如果偏自相关在第一次延迟后消失，那么就应该基于自身的单延迟（一个时间段的延迟）版本对序列建模。然而，如果偏自相关在第一次延时后没有消失，那么就需要在自回归模型中使用时间序列的多个延时版本。

可以设想，如果用连续增大的延迟值（x_{t-1}，x_{t-2} 等）为序列中某个值（x_t）建立线性模型。等式可能如下所示：

$$x_t=m_1x_{t-1}+m_2x_{t-2}+m_3x_{t-3}+\cdots+b$$

系数 m_1、m_2 等分别是一个时间段延迟的偏自相关、两个时间段延迟的偏自相关，等等。因此，为了计算某个延迟的偏自相关，需要做的是估算给出相应系数的线性回归公式。执行此计算的函数称为**偏自相关函数**（PACF）。

使用我们最喜欢的线性回归软件包 `github.com/sajari/regression`，可以创建实现 PACF 的 Go 函数，如下所示：

```
// pacf calculates the partial autocorrelation for a series
// at the given lag.
func pacf(x []float64, lag int) float64 {

    // Create a regresssion.Regression value needed to train
    // a model using github.com/sajari/regression.
    var r regression.Regression
    r.SetObserved("x")

    // Define the current lag and all of the intermediate lags.
    for i := 0; i < lag; i++ {
        r.SetVar(i, "x"+strconv.Itoa(i))
    }
```

仅供非商业用途或交流学习使用


```
    // Shift the series.
    xAdj := x[lag:len(x)]

    // Loop over the series creating the data set
    // for the regression.
    for i, xVal := range xAdj {

        // Loop over the intermediate lags to build up
        // our independent variables.
        laggedVariables := make([]float64, lag)
        for idx := 1; idx <= lag; idx++ {

            // Get the lagged series variables.
            laggedVariables[idx-1] = x[lag+i-idx]
        }

        // Add these points to the regression value.
        r.Train(regression.DataPoint(xVal, laggedVariables))
    }

    // Fit the regression.
    r.Run()

    return r.Coeff(lag)
}
```

然后，可以使用这个 pacf 函数来计算偏自相关的一些值，类似于之前计算的自相关的延迟，如下所示：

```
// Open the CSV file.
passengersFile, err := os.Open("AirPassengers.csv")
if err != nil {
    log.Fatal(err)
}
defer passengersFile.Close()

// Create a dataframe from the CSV file.
passengersDF := dataframe.ReadCSV(passengersFile)

// Get the time and passengers as a slice of floats.
passengers := passengersDF.Col("AirPassengers").Float()

// Loop over various values of lag in the series.
fmt.Println("Partial Autocorrelation:")
for i := 1; i < 11; i++ {

    // Calculate the partial autocorrelation.
    pac := pacf(passengers, i)
    fmt.Printf("Lag %d period: %0.2f\n", i, pac)
}
```

编译并运行这个程序将得到航空乘客时间序列的偏自相关：

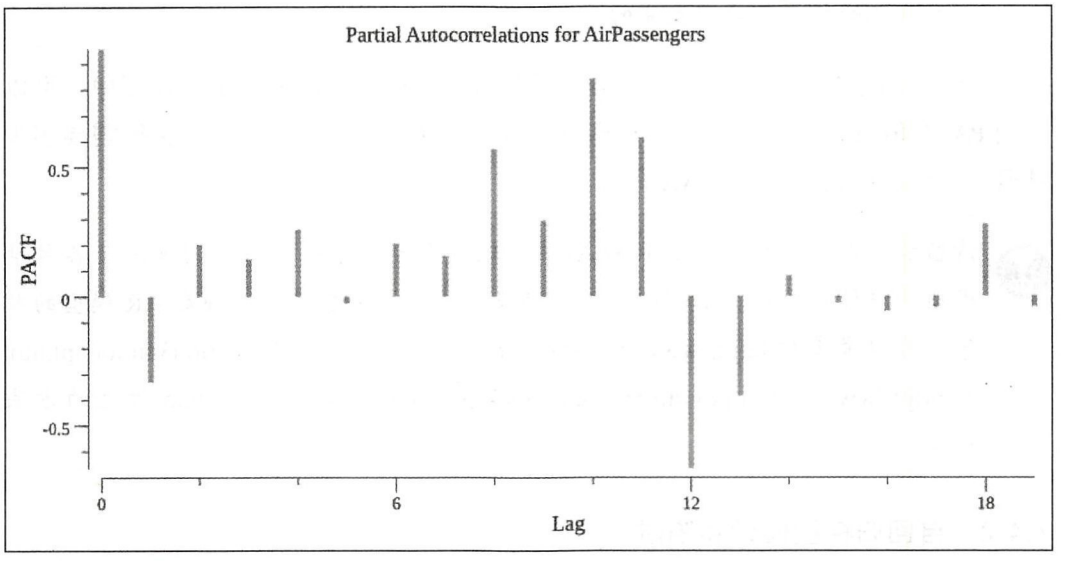
```
$ go build
$ ./myprogram
Partial Autocorrelation:
Lag 1 period: 0.96
Lag 2 period: -0.33
Lag 3 period: 0.20
```

仅供非商业用途或交流学习使用


```
Lag 4 period: 0.15
Lag 5 period: 0.26
Lag 6 period: -0.03
Lag 7 period: 0.20
Lag 8 period: 0.16
Lag 9 period: 0.57
Lag 10 period: 0.29
```

如你所见，偏自相关在第二次延迟后迅速消失。这表明在考虑时间序列与其第一次和第二次延迟之间的关系后，在时间序列中没有太多剩余关系。偏自相关并没有精确达到 *0.0*，是因为数据中存在一些噪声，所以这是预期的。

为了使 PACF 更加直观，让我们创建另一个图。可以像之前 ACF 那样操作，只需将 `acf()` 函数替换为 `pacf()` 函数。结果图如下所示：



7.4　预测的自回归模型

用来预测时间序列的第一类模型称为**自回归**（AR）模型。如前所述，将尝试基于序列中一个或多个之前的点，对时间序列中的数据点进行建模。也就是说使用时间序列本身对时间序列进行建模。对序列自身的使用使 AR 方法有别于第 4 章中讨论的更一般的方法。

7.4.1　自回归模型概述

我们经常会看到称为 AR(1)、AR(2) 等的 AR 模型。这些数字对应于用来建模时间序列的 AR 模型或过程的**顺序**。可以根据这个顺序决定执行自相关和偏自相关分析。

仅供非商业用途或交流学习使用

AR(1) 模型试图根据同一序列在一个时间延迟观察的基础上，对序列中的观察进行建模：

$$x_t=m_1x_{t-1}+b$$

AR(2) 模型看起来如下所示：

$$x_t=m_1x_{t-1}+m_2x_{t-2}+b$$

AR(3) 模型会添加另一个多项式的项，以此类推都遵循这种模式。

这些公式可能让你想到线性回归，实际上这里用到的许多方法同创建线性回归模型时所用的方法相同。但是，时间序列建模的独到之处在这里也很重要。对数据中时间的相关要素（季节性、趋势、自相关等）以及它们如何影响 AR 模型有直观的认识是非常重要的。

创建时间序列 AR 模型的顺序最好通过查看 PACF 图形来决定。在图中，可以看到 PACF 值衰减，然后在零附近徘徊。看看 PACF 开始在零点附近徘徊需要多少延迟，然后利用相应延时的 AR 命令。

请留意一些用于绘制 PACF 和 ACF 的软件会包含水平线段，用来表示各种延迟项的统计重要性。本书此处并未涉及，但是如果想定量地确定 AR 模型的顺序，可以考虑按 http://www.itl.nist.gov/div898/handbook/eda/section3/autocopl.htm 和 http://www.itl.nist.gov/div898/handbook/pmc/section4/pmc4463.htm 所述的方法计算。

7.4.2 自回归模型假设和陷阱

自回归模型的主要假设如下。

- **平稳性**：AR 模型假定时间序列是平稳的。如果打算使用 AR 模型，就不应该在数据中看到任何趋势。
- **遍历性**：这个独特的术语基本上意味着时间序列的统计特征，如均值和方差，不应随时间变化或漂移。

使用 AR 方法建模的任何时间序列都应该满足这些假设。然而，即使一些数据（如航空乘客数据）不符合这些假设，仍可以使用一些差异技巧来利用 AR 模型。

7.4.3 自回归模型示例

这里将尝试使用自回归模型来对航空旅客数据建模。现在已经知道正在打破 AR 模型的一个假设，因为数据并不稳定。但是，可以应用一个常见的技巧来使序列保

仅供非商业用途或交流学习使用

持稳定，这个技巧称为**差分**。

7.4.3.1　转变为一个稳定的序列

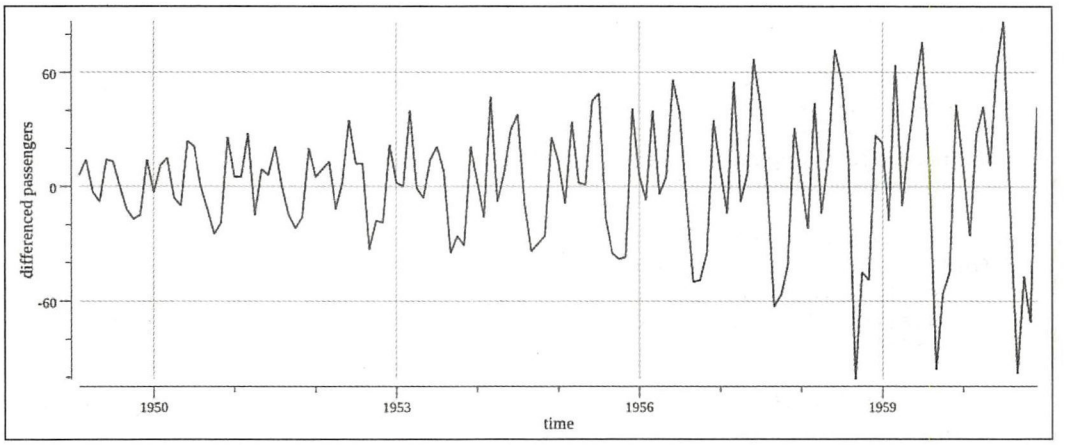
为了让时间序列保持平稳，将创建一个代理时间序列，其中时间段 t 的观察值，是原始序列中时间段 t 的观察值减去前一次观察值。以这种方式差分每个观察结果，然后绘制结果以查看它是否摆脱了数据中的趋势。也将输出这个差分的时间序列到一个新的 `*.csv` 文件，如下面的代码所示：

```
// as slices of floats.
passengerVals := passengersDF.Col("AirPassengers").Float()
timeVals := passengersDF.Col("time").Float()

// pts will hold the values for plotting.
pts := make(plotter.XYs, passengersDF.Nrow()-1)

// differenced will hold our differenced values
// that will be output to a new CSV file.
var differenced [][]string
differenced = append(differenced, []string{"time",
"differenced_passengers"})

// Fill pts with data.
for i := 1; i < len(passengerVals); i++ {
    pts[i-1].X = timeVals[i]
    pts[i-1].Y = passengerVals[i] - passengerVals[i-1]
    differenced = append(differenced, []string{
        strconv.FormatFloat(timeVals[i], 'f', -1, 64),
        strconv.FormatFloat(passengerVals[i]-passengerVals[i-1], 'f', -1,
64),
    })
}

// Create the plot.
p, err := plot.New()
if err != nil {
    log.Fatal(err)
}
p.X.Label.Text = "time"
p.Y.Label.Text = "differenced passengers"
p.Add(plotter.NewGrid())

// Add the line plot points for the time series.
l, err := plotter.NewLine(pts)
if err != nil {
    log.Fatal(err)
}
l.LineStyle.Width = vg.Points(1)
l.LineStyle.Color = color.RGBA{B: 255, A: 255}

// Save the plot to a PNG file.
p.Add(l)
if err := p.Save(10*vg.Inch, 4*vg.Inch, "diff_passengers_ts.png"); err !=
nil {
    log.Fatal(err)
}
```

仅供非商业用途或交流学习使用

```
// Save the differenced data out to a new CSV.
f, err := os.Create("diff_series.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

w := csv.NewWriter(f)
w.WriteAll(differenced)

if err := w.Error(); err != nil {
    log.Fatal(err)
}
```

结果如下图所示：



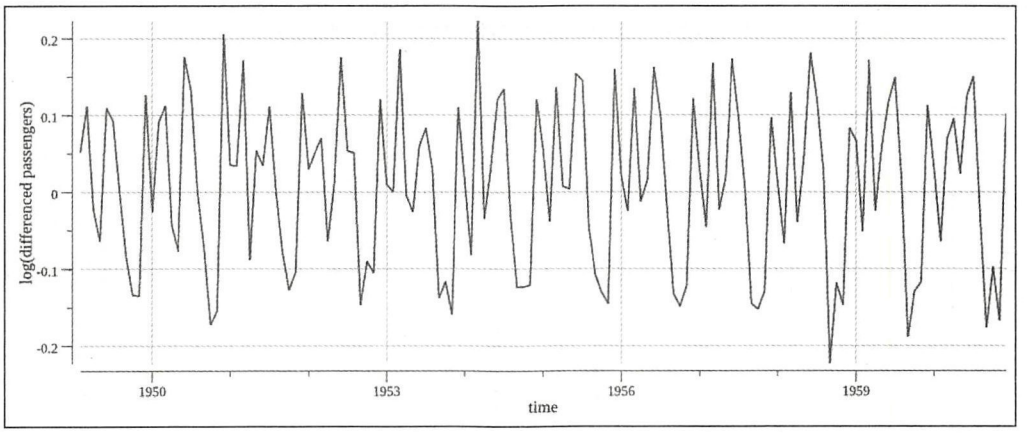
在这里，可以看到，已经基本消除了原始时间序列中上升趋势的所有迹象。但是，似乎仍然存在与方差有关的问题。随着时间推移，差分时间序列与均值的方差看起来在增大，这打破了遍历性假设。

为了应对方差的增长，可以利用对数（log）或者幂（power）转换进一步转换时序，以稍后惩罚时序中的较大值。添加此对数转换，重新绘制差分序列的对数图表，然后将结果数据保存到名为 `log_diff_series.csv` 的文件中。完成此操作的代码与前面的代码片段相同，只是使用 `math.Log()` 来转换每个值，因此将省去细节。以下是结果图：

真棒！现在有了一个看起来是可以在 AR 模型中使用的平稳序列。请注意，虽然在这个例子中大部分都是定性的，但是有平稳性的定量测试（例如 Dickey-Fuller 测试）。

在这里用差分和对数转换来转换数据，使得序列能够符合 AR 模型的假设，但它也使数据和最终模型不太容易理解。与时间序列本身相比，考虑差分时间序列的对数更困难。在这里做折中是有理由的，这个折中应该记录下来，并且期望能够避免这种困惑。

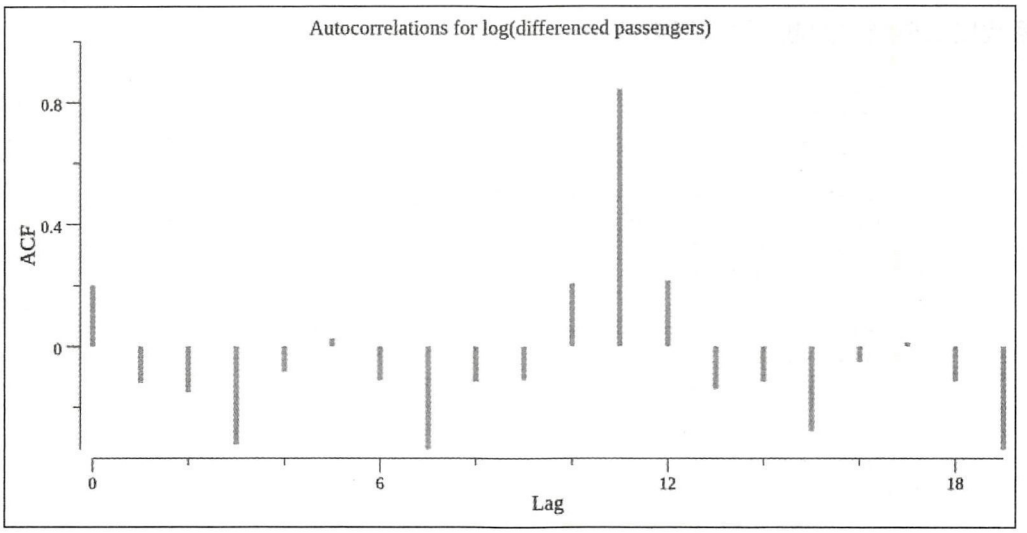
仅供非商业用途或交流学习使用



7.4.3.2　分析 ACF 并选择 AR 顺序

现在有一个平稳的序列，符合模型的假设，请重新审视 ACF 和 PACF 图，看看发生了什么变化。可以使用之前绘制 ACF 和 PACF 的相同代码，但是这次将使用变换后的序列。

这是由此产生的 ACF 图：

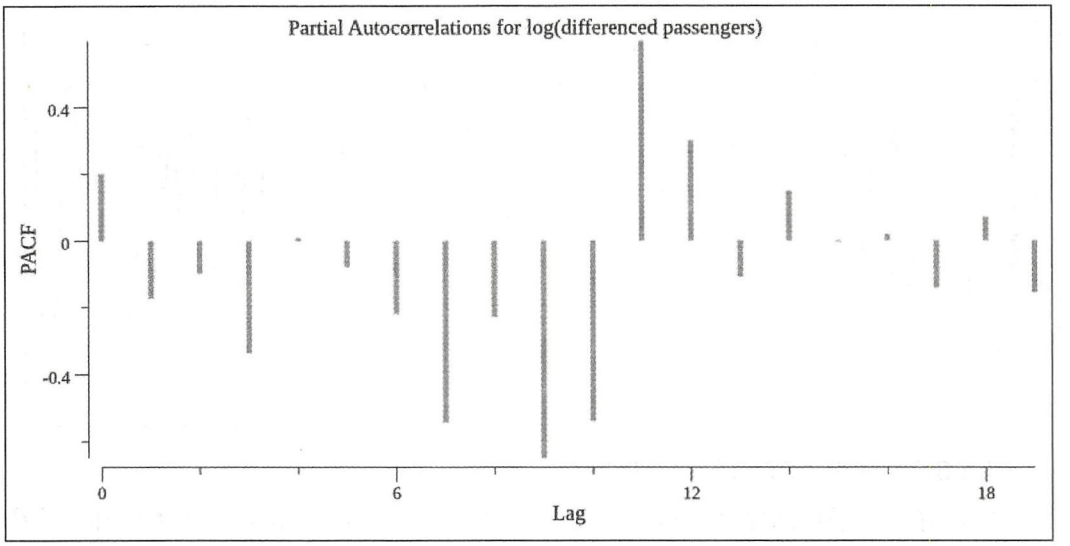


这是由此产生的 PACF 图：

第一眼看到的就是随着延迟越来越大，ACF 图不再从 *1.0* 缓慢衰减。ACF 图衰减并在 *0.0* 附近波动。在本章后续部分将再次回到 ACF。

接下来，可以看到，PACF 也衰减到 *0.0*，然后在 *0.0* 附近波动。为了选择 AR 模型的顺序，需要检查 PACF 曲线看起来第一次穿过零线的位置。在这个例子中，似乎是在第二个延迟期之后，因此，可能想要考虑使用 AR(2) 模型来自动回归模拟这个时间序列。

仅供非商业用途或交流学习使用



7.4.3.3 拟合和评估 AR(2) 模型

可以看到，PACF 提供了 AR 模型中各种顺序的系数。利用这一点，可以使用下面的代码中所示的稍微修改过的 `pacf()` 函数版本，获得第一个和第二个延迟项的系数以及模型中的截距（或误差项）：

```
// autoregressive calculates an AR model for a series
// at a given order.
func autoregressive(x []float64, lag int) ([]float64, float64) {

    // Create a regresssion.Regression value needed to train
    // a model using github.com/sajari/regression.
    var r regression.Regression
    r.SetObserved("x")

    // Define the current lag and all of the intermediate lag
    for i := 0; i < lag; i++ {
        r.SetVar(i, "x"+strconv.Itoa(i))
    }

    // Shift the series.
    xAdj := x[lag:len(x)]

    // Loop over the series creating the data set
    // for the regression.
    for i, xVal := range xAdj {

        // Loop over the intermediate lags to build up
        // our independent variables.
        laggedVariables := make([]float64, lag)
        for idx := 1; idx <= lag; idx++ {

            // Get the lagged series variables.
            laggedVariables[idx-1] = x[lag+i-idx]
        }
```

仅供非商业用途或交流学习使用

```
            // Add these points to the regression value.
            r.Train(regression.DataPoint(xVal, laggedVariables))
        }

        // Fit the regression.
        r.Run()

        // coeff hold the coefficients for our lags.
        var coeff []float64
        for i := 1; i <= lag; i++ {
            coeff = append(coeff, r.Coeff(i))
        }

        return coeff, r.Coeff(0)
    }
```

然后，可以在对数差分序列上调用它来获得训练的 AR(2) 模型系数：

```
// Open the CSV file.
passengersFile, err := os.Open("log_diff_series.csv")
if err != nil {
    log.Fatal(err)
}

defer passengersFile.Close()
// Create a dataframe from the CSV file.
passengersDF := dataframe.ReadCSV(passengersFile)

// Get the time and passengers as a slice of floats.
passengers := passengersDF.Col("log_differenced_passengers").Float()

// Calculate the coefficients for lag 1 and 2 and
// our error.
coeffs, intercept := autoregressive(passengers, 2)

// Output the AR(2) model to stdout.
fmt.Printf("\nlog(x(t)) - log(x(t-1)) = %0.6f + lag1*%0.6f +
lag2*%0.6f\n\n", intercept, coeffs[0], coeffs[1])
```

编译并运行此训练程序，可提供以下 AR(2) 公式，以计算不同乘客数量的对数：

```
$ go build
$ ./myprogram

log(x(t)) - log(x(t-1)) = 0.008159 + lag1*0.234953 + lag2*-0.173682
```

为了评估这个 AR(2) 模型，可以计算**平均绝对误差**（MAE），类似于为线性回归模型计算 MAE。具体来说，将计算与观察的乘客数量配对的预测的乘客数量，然后计算误差并累计 MAE。

首先，来计算转换后的（对数和差分）预测：

```
// Open the log differenced dataset file.
transFile, err := os.Open("log_diff_series.csv")
if err != nil {
    log.Fatal(err)
```

仅供非商业用途或交流学习使用

```
}
defer transFile.Close()

// Create a CSV reader reading from the opened file.
transReader := csv.NewReader(transFile)

// Read in all of the CSV records
transReader.FieldsPerRecord = 2
transData, err := transReader.ReadAll()
if err != nil {
    log.Fatal(err)
}

// Loop over the data predicting the transformed
// observations.
var transPredictions []float64
for i, _ := range transData {

    // Skip the header and the first two observations
    // (because we need two lags to make a prediction).
    if i == 0 || i == 1 || i == 2 {
        continue
    }

    // Parse the first lag.
    lagOne, err := strconv.ParseFloat(transData[i-1][1], 64)
    if err != nil {
        log.Fatal(err)
    }

    // Parse the second lag.
    lagTwo, err := strconv.ParseFloat(transData[i-2][1], 64)
    if err != nil {
        log.Fatal(err)
    }

    // Predict the transformed variable with our trained AR model.
    transPredictions = append(transPredictions,
0.008159+0.234953*lagOne-0.173682*lagTwo)
}
```

现在，为了计算 MAE，需要将这些预测值转换回正常的乘客数量（这样可以直接将它们与原始时间序列进行比较）。对数和差分数据的逆向变换包括，在转换后的序列中累计求和，然后将它们加回到基本序列值中，再然后取指数。这种逆向变换，MAE 的累计，以及绘制观察和预测点的集合如下所示：

```
// Open the original dataset file.
origFile, err := os.Open("AirPassengers.csv")
if err != nil {
    log.Fatal(err)
}
defer origFile.Close()

// Create a CSV reader reading from the opened file.
origReader := csv.NewReader(origFile)
```

仅供非商业用途或交流学习使用

```
// Read in all of the CSV records
origReader.FieldsPerRecord = 2
origData, err := origReader.ReadAll()
if err != nil {
    log.Fatal(err)
}

// pts* will hold the values for plotting.
ptsObs := make(plotter.XYs, len(transPredictions))
ptsPred := make(plotter.XYs, len(transPredictions))

// Reverse the transformation and calculate the MAE.
var mAE float64
var cumSum float64
for i := 4; i <= len(origData)-1; i++ {

    // Parse the original observation.
    observed, err := strconv.ParseFloat(origData[i][1], 64)
    if err != nil {
        log.Fatal(err)
    }

    // Parse the original date.
    date, err := strconv.ParseFloat(origData[i][0], 64)
    if err != nil {
        log.Fatal(err)
    }

    // Get the cumulative sum up to the index in
    // the transformed predictions.
    cumSum += transPredictions[i-4]

    // Calculate the reverse transformed prediction.
    predicted := math.Exp(math.Log(observed) + cumSum)

    // Accumulate the MAE.
    mAE += math.Abs(observed-predicted) / float64(len(transPredictions))

    // Fill in the points for plotting.
    ptsObs[i-4].X = date
    ptsPred[i-4].X = date
    ptsObs[i-4].Y = observed
    ptsPred[i-4].Y = predicted
}
```

然后，输出 MAE 以突出显示并保存观察值和预测值的线图：

```
// Output the MAE to standard out.
fmt.Printf("\nMAE = %0.2f\n\n", mAE)

// Create the plot.
p, err := plot.New()
if err != nil {
    log.Fatal(err)
}
p.X.Label.Text = "time"
p.Y.Label.Text = "passengers"
p.Add(plotter.NewGrid())
```

仅供非商业用途或交流学习使用

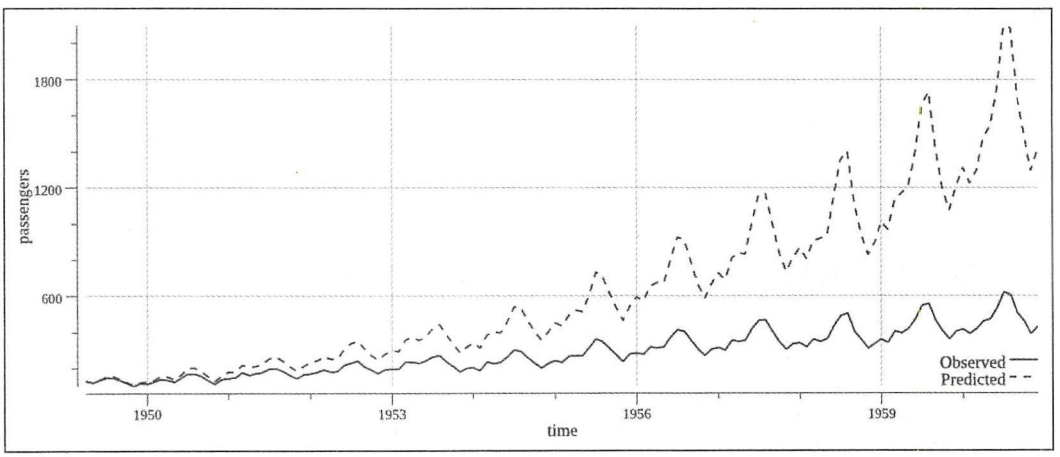
仅供非商业用途或交流学习使用


```
// Add the line plot points for the time series.
lObs, err := plotter.NewLine(ptsObs)
if err != nil {
    log.Fatal(err)
}
lObs.LineStyle.Width = vg.Points(1)

lPred, err := plotter.NewLine(ptsPred)
if err != nil {
    log.Fatal(err)
}
lPred.LineStyle.Width = vg.Points(1)
lPred.LineStyle.Dashes = []vg.Length{vg.Points(5), vg.Points(5)}

// Save the plot to a PNG file.
p.Add(lObs, lPred)
p.Legend.Add("Observed", lObs)
p.Legend.Add("Predicted", lPred)
if err := p.Save(10*vg.Inch, 4*vg.Inch, "passengers_ts.png"); err != nil {
    log.Fatal(err)
}
```

编译并运行它将获得以下 MAE：

```
$ go build
$ ./myprogram

MAE = 355.20
```

回想一下这个序列最初的可视化，乘客的数量从零到略高于 600。因此，大约 355 时的 MAE 并不是很好。然而，为了更全面地了解预测和观察结果，来看看前面代码所生成的图：



如图所示，这个模型过度预测了乘客的数量，特别是随着时间的推移。该模型确实展现了原始数据中可见的一些结构，并产生了类似的趋势。然而，似乎需要一个稍微复杂一些的模型来更真实地表示这个序列。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

没有模型是完美的，在这里尝试了一个相对简单的时间序列模型。试图坚持一个简单而可解释的模型是好事，但评估结果可能会激励我们在现实场景中重构模型。重构是好的！这意味着学到了新的内容。

7.5　自回归移动平均和其他时间序列模型

之前尝试的模型是一个相对简单的纯自回归模型。但是，并不是要坚持说在时间序列模型中单独使用自回归或纯自回归。正如本书涵盖的其他类型的机器学习模型一样，有大量的时间序列技术可以使用，在这里无法一一介绍它们。但是，确实希望提到一些值得关注的技术，可以让读者在阅读本书时探索这些技术。

自回归模型通常与称为**移动平均模型**（moving average model）的模型相结合。当这些结合起来时，它们通常被称为**自回归移动平均**（ARMA，Auto-Regressive Moving Average）或**差分自回归移动平均**（ARIMA，Auto-Regressive Integrated Moving Average）模型。ARMA/ARIMA 模型的移动平均部分允许在时间序列中捕捉诸如白噪声或其他误差项，这会实际上改善 AR(2) 航空乘客模型。

不幸的是，在编写这些内容时，没有开箱即用的软件包来执行 Go 中的 ARIMA。如前所述，自回归部分相对容易，但移动平均拟合稍微复杂一些。这是另一个吸引人的好地方！

还有一些时间序列模型不属于 ARIMA 模型领域。例如，Holt-Winters 方法试图通过预测方程和三个平滑方程来捕获时间序列数据的季节性。在 `github.com/datastream/holtwinters` 和 `github.com/dgryski/go-holtwinters` 中有 Holt-Winters 方法的初步实现。这些可能需要进一步维护和产品化，但它们提供了一个起点。

7.6　异常检测

正如本章导言所提到的，预测时间序列并不能让人一直保持兴趣。我们可能想要在时间序列中检测异常行为。例如，可能想知道什么时候网络中会发生不寻常的突发流量，或者当超出一般用户数量的用户在应用程序内尝试某些事情时，这些异常操作需要发出警报。这些事件可能与安全问题有关，也可能仅用于调整基础架构或应用程序设置。

值得庆幸的是，由于 Go 在监控和基础架构方面的使用历史，Go 有多种选项可

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

用于检测时间序列数据中的异常情况。这种工具已被用于生产过程中，用于检测异常行为，同时监控基础设施和应用程序。尽管有很多工具无法一一提及，但会重点介绍其中的两个。

首先，InfluxDB（https://www.influxdata.com/）和 Prometheus（https://prometheus.io/）生态系统有多种异常检测选项。InfluxDB 和 Prometheus 都提供开源的基于 Go 的时间序列数据库和相关工具。它们对于监控基础架构和应用程序都非常有用，并且它们在 Go 社区和 Go 社区之外都有广泛使用。例如，如果你有兴趣使用 InfluxDB，则可以使用 `github.com/nathanielc/morgoth` 进行异常检测。

这个包实现了**有损计数算法**（LCA，Lossy Counting Algorithm）。在 Prometheus 方面，可以利用基于查询的方法，详情请见 https://prometheus.io/blog/2015/06/18/practical-anomaly-detection/。

还有各种独立 Go 包可用于异常检测，包括 `github.com/lytics/anomalyzer` 和 `github.com/sec51/goanomaly`。具体来说，`github.com/lytics/anomalyzer` 执行各种测试来确定序列中的观察结果是否异常，包括基于累积分布函数、自举排列、置换排序和相对量级等的测试。

要使用 `github.com/lytics/anomalyzer` 检测异常，需要创建一些配置和 `anomalyzer.Anomalyzer` 值。一旦完成了这一步，检测异常就像调用 `anomalyzer.Anomalyzer` 上的 `Push()` 方法一样简单，如下面的代码所示：

```
// Initialize an AnomalyzerConf value with
// configurations such as which anomaly detection
// methods we want to use.
conf := &anomalyzer.AnomalyzerConf{
    Sensitivity: 0.1,
    UpperBound:  5,
    LowerBound:  anomalyzer.NA, // ignore the lower bound
    ActiveSize:  1,
    NSeasons:    4,
    Methods:     []string{"diff", "fence", "highrank", "lowrank",
"magnitude"},
}

// Create a time series of periodic observations
// as a slice of floats.  This could come from a
// database or file, as utilized in earlier examples.
ts := []float64{0.1, 0.2, 0.5, 0.12, 0.38, 0.9, 0.74}

// Create a new anomalyzer based on the existing
// time series values and configuration.
anom, err := anomalyzer.NewAnomalyzer(conf, ts)
if err != nil {
    log.Fatal(err)
}
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
// Supply a new observed value to the Anomalyzer.
// The Anomalyzer will analyze the value in reference
// to pre-existing values in the series and output
// a probability of the value being anomalous.
prob := anom.Push(15.2)
fmt.Printf("Probability of 15.2 being anomalous: %0.2f\n", prob)

prob = anom.Push(0.43)
fmt.Printf("Probability of 0.33 being anomalous: %0.2f\n", prob)
```

编译并运行这个异常检测会得到以下结果：

```
$ go build
$ ./myprogram
Probability of 15.2 being anomalous: 0.98
Probability of 0.33 being anomalous: 0.80
```

7.7　参考书目

时间序列统计（ACF 和 PACF）：

- 如何使用 ACF：https://coolstatsblog.com/2013/08/07/how-to-use-the-autocorreation-function-acf/
- 识别 ARIMA 模型中 AR 或 MA 项的数量：https://people.duke.edu/~rnau/411arim3.htm

自回归模型：

- AR 模型的更多数学介绍：https://onlinecourses.science.psu.edu/stat501/node/358
- `github.com/sajari/regression` 文档：https://godoc.org/github.com/sajari/regression

ARMA/ARIMA 模型：

- ARIMA 简介：https://people.duke.edu/~rnau/411arim.htm

异常检测：

- InfluxDB：https://www.influxdata.com/
- Prometheus：https://prometheus.io/
- `github.com/lytics/anomalyzer` 文档：https://godoc.org/github.com/lytics/anomalyzer
- `github.com/sec51/goanomaly` 文档：https://godoc.org/github.com/sec51/goanomaly

仅供非商业用途或交流学习使用

7.8 小结

这非常及时！现在知道什么是时间序列数据，如何在 Go 中表示它，如何进行一些预测，以及如何检测时间序列数据中的异常情况。只要是处理随时间变化的数据，无论数据是与股票价格相关的数据，还是监控与基础设施相关的数据，这些技能都将非常有用。

在下一章中，将通过研究一些先进技术（包括神经网络和深度学习）来提高基于 Go 语言的机器学习的水平。

仅供非商业用途或交流学习使用

第 8 章
神经网络和深度学习

在本书中，已经探讨了一些用来预测的机器训练和培训。为此采用了各种各样有用和有趣的算法，其中包括各种类型的回归、决策树、最近邻居。但是让我们退一步想想，如果希望做出准确预测并了解数据，那么想要模拟现实中的什么呢？

这个问题的答案显而易见，模拟人脑。作为人类，天生具有识别物体、预测数量、识别欺诈等能力。而这些正是我们希望机器可以完成的。当然，我们在这些活动中并不完美，但做得还不错。

这种思维方式导致了**人工神经网络**（也被称为**神经网络**或**神经网**）的发展。这些模型尝试大致模拟人脑的某些结构 ，例如**神经元**。它们在各个行业中都取得了广泛成功，现在被应用于解决各种有趣的问题。

最近，很多专业和复杂的神经网络类型吸引了人们大量的兴趣和注意。这些神经网络属于**深度学习**的范畴，通常具有比所谓常规神经网络**更深的**结构。也就是说，它们具有很多隐藏的结构层，并且能被数千万个参数或权重参数化。

本章将尝试介绍基于 Go 语言的神经网络和深度学习模型。这个话题非常广泛，关于深度学习就有大量的书籍。因此这里仅能了解到皮毛。话虽如此，以下的内容应该能为使用 Go 语言构建神经网络提供坚实的起点。

8.1 理解神经网络术语

神经网络的类别非常多，每种类别都有一套自己的术语。但是无论使用哪种类型的神经网络，都应该知道一些常见的术语。这些术语如下所示：

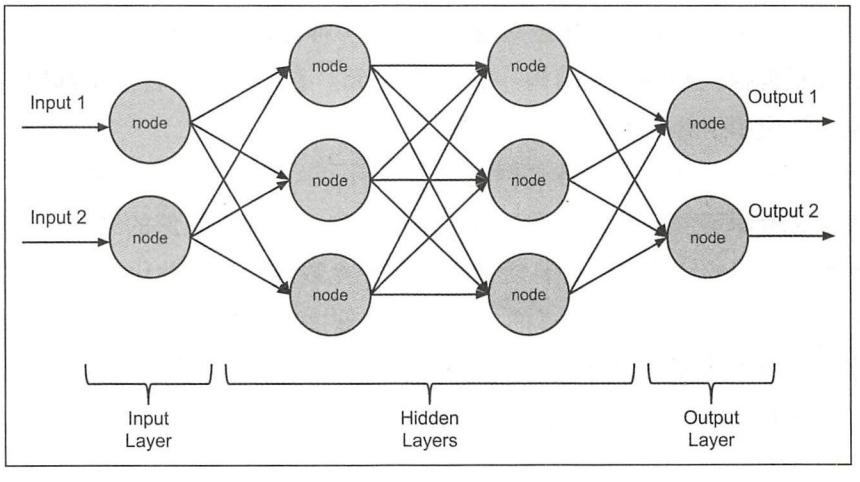
- **节点**、**感知器**或**神经元**：这些术语都表示相同的意思，它们指的是神经网络

仅供非商业用途或交流学习使用

的基本组成部分。每个节点或神经元接收输入数据并对该数据执行操作。执行完操作后，该节点 / 神经元可能会或不会传递操作到下一个节点 / 神经元。

- **激活**：与操作节点相关的输出或者值。
- **激活函数**：将节点输入转化成输出或激活操作的函数。
- **权重**或**偏差**：这些值定义了激活函数中输入和输出数据之间的关系。
- **输入层**：神经网络的输入层包括一系列节点，用于接收神经网络模型的初始化输入（例如一系列的特征或属性）。
- **输出层**：神经网络的输出层包括一系列节点，用于将信息传递到神经网络内部，并将其转化为最终输出。
- **隐藏层**：神经网络的这些层存在于输入层和输出层之间。因此对于外部输入或输出的**隐藏**的。
- **前馈**或**正向传输**：这是指将数据输入到神经网络的输入层，并正向传输到输出层（没有任何循环）。
- **反向传播**：这是一种训练神经网络模型的方法，它涉及通过网络提供前馈数值，计算生成的误差，然后将基于这些误差的更改传回到网络。
- **架构**：神经元如何在神经网络中连接在一起的整体结构称为架构。

为了巩固这些概念，请考虑如下神经网络的示意图：



这是一个基本的前馈（即非循环或递归）神经网络。它有两个隐藏层，接收两个输入，并输出两个类值（或结果）

如果这些数据有些难以理解，请不用担心。接下来将看到一些具体的例子，用以巩固所有这些零散的知识。如果因本章中各种示例而焦头烂额甚至混淆了术语。请回到这里复查。

仅供非商业用途或交流学习使用

8.2　构建一个简单的神经网络

许多神经网络软件包和神经网络应用将模型视为黑盒子。也就是说，人们倾向于使用一些框架，以便使用一系列默认值和自动化来快速构建神经网络。这种操作往往能产生一些结果，但这种便利不会对模型如何工作建立直觉。因此，当这些模型行为和预期不一致时，很难理解为什么模型会做出奇怪的预测或很难收敛。

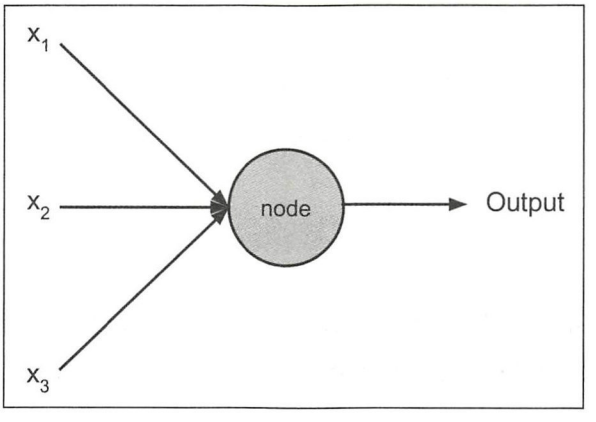
在进入更复杂的神经网络学习之前，让我们建立一些关于神经网络的基本直觉，以免陷入这些问题无法自拔。这里将从头开始创建一个简单的神经网络，用以了解神经网络的基本组成，以及它们如何一起运作。

注意：即使要从头开始构建神经网络（某些情况下你可能也想这么做），也可以利用现有的许多 Go 软件包，它们有助于神经网络的构建、训练和预测。其中包括 github.com/tleyden/neurgo、github.com/fxsjy/gonn、github.com/NOX73/go-neural、github.com/milosgajdos83/gosom、github.com/made2591/go-perceptron-go 及 github.com/chewxy/gorgonia。

本章将利用神经网络进行分类。然而，神经网络也可用来执行回归。在这里可以了解更多关于该主题的信息：https://heuristically.wordpress.com/2011/11/17/using-neural-network-for- regression/。

8.2.1　网络中的节点

神经网络中的节点或神经元本身具有相对简单的功能。每个神经元都会取一个或多个值（x_1，x_2 等）。根据一个激活函数组合这些值，并产生单个输出。以下是图形化的输出。



仅供非商业用途或交流学习使用

究竟如何组合输入才能得到输出呢？需要一种方法来组合可调整的输入（以便能训练模型）。可以看到，使用系数和截距的组合变量，是组合输入的一种可训练方式。回想一下第 4 章。本着这种思路，将输入线性地与一些系数（**权重**）和一个截距（**偏差**）结合起来：

$$w_1x_1+w_2x_2+w_3x_3+b$$

这里，w_1，w_2 等是权重，b 是偏差。

输入的这种组合是一个很好的开始，但是它是线性的，因此无法在输入和输出之间创建非线性关系的模型。为了引入一下非线性，将对这种线性输入的组合应用激活函数。这里使用的激活函数类与第 5 章中介绍的逻辑函数类似。在神经网络和如下形式中，逻辑函数被称为 `sigmoid` 函数：

$$sigmoid(x)=\frac{1}{1+e^{-x}}$$

在节点中使用 `sigmoid` 函数是一个很好的选择，因为它引入了非线性，但是它也具有有限的取值范围（介于 *0* 和 *1* 之间），具有简单定义的导数（将在网络训练时候用到），同时它还有概率解释（如第 5 章中进一步讨论的）。

Sigmoid 函数并不是神经网络中激活函数的唯一选择。其他流行的选择还有双曲线正切函数、softmax 和整流线性函数。激活函数的选择以及它们的优缺点在这里有进一步讨论：https://medium.com/towards-data-science/activation-functions-and-its-types-which-is-better-a9a5310cc8f。

让我们继续，在 Go 中定义激活函数及其导数。这些定义如下面代码所示：

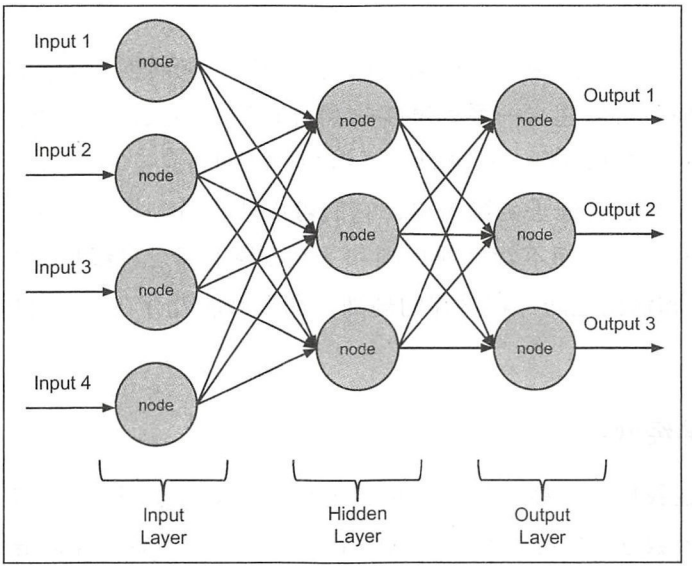
```
// sigmoid implements the sigmoid function
// for use in activation functions.
func sigmoid(x float64) float64 {
    return 1.0 / (1.0 + math.Exp(-x))
}

// sigmoidPrime implements the derivative
// of the sigmoid function for backpropagation.
func sigmoidPrime(x float64) float64 {
    return x * (1.0 - x)
}
```

8.2.2 网络架构

要构建的简单神经网络包含一个输入层和一个输出层（就像任何神经网络一样）。网络将在输入层和输出层之间包含一个隐藏层。该架构如下所示：

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用



需要特别指出的是，这里将包含输入层中的 4 个节点、隐藏层中的 3 个节点以及输出层中的 3 个节点。输入层的 4 个节点对应将要反馈到网络中的属性数量。将这 4 个输入想象成在第 5 章中用于分类鸢尾花的 4 个测量值。输出层包含 3 个节点，这里将搭建用于鸢尾花分类的网络，因此它可能是鸢尾花的 3 个类别之一。

现在探讨一下隐藏层问题。为什么使用有 3 个节点的隐藏层？对于绝大多数简单任务而言，一个隐藏层就足够了。如果数据中存在大量的非线性、多输入并且 / 或者有大量的训练数据，则可能需要引入更多的隐藏层（本章稍后在有关深度学习的地方将进一步讨论）。隐藏层中的 3 个节点可以根据评估指标和一些试验与误差进行调整。还可以在隐藏层中搜索节点以便自动选择。

请留意，隐藏层中引入的节点越多，网络就能越完美地从训练集中学习。换句话说，会置模型于过度拟合的风险中，以致于无法一般化。当添加这种复杂化特性时，仔细验证模型确保其一般化是非常重要的。

8.2.3　为什么期望这种架构有作用

让我们暂停一下，然后想想为什么在前面安排中设置一系列节点就可以预测一些东西？如你所见，这里做的是不断合并输入用以产生一些结果。如何期望这个结果对二元分类有用？

当定义一个二元分类问题或任何分类问题或回归问题时，究竟意味着什么？基本上来说，期望在一系列输入（属性）和一个输出（标签或响应）之间存在一些规则或关系。实际上，也就是说存在一些简单或复杂的函数，可以将属性转换为之后的

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

响应或标签。

可以选择一种类型的函数来模拟这种属性到输出的转换，就像线性回归或者逻辑回归那样。但也可以建立一系列链接的和可调整的函数，通过算法训练来检测输入和输出之间的关系和规则。这就是正在做的神经网络。

神经网络的节点就像子函数一样，它们可以调整直到模拟出所提供的输入和输出之间的规则和关系，而不管这些规则和关系实际是什么（线性、非线性、动态的等）。如果输入和输出之间存在潜在的规则，那么可能就存在一些可以模拟这些规则的神经网络。

8.2.4 训练神经网络

好的，现在我们有一些很好的动机，为什么这个节点组合可以有助于预测。如何根据一些输入数据，实际调整神经网络节点的所有子功能？答案是**反向传播**。

反向传播是一种训练神经网络的方法，它涉及在一系列**阶段**（或暴露给训练数据集）迭代地进行以下操作。

- 通过神经网络前馈训练数据用以计算输出
- 计算输出中的误差
- 使用梯度下降法（或其他方法）来确定如何根据误差来更改权重和偏差
- 将这些权重 / 偏差的变化反向传播到网络中

稍后将实现这个过程，并检查一些细节。在附录中将详细介绍梯度下降等内容。

这种用于训练神经网络的反向传播的方法在建模世界中无处不在，还有大量的网络架构和方法本书无法一一涵盖。神经网络是学术界和工业界活跃的研究领域。

首先定义一个 `neuralNetConfig` 结构体，它定义了网络架构的参数，以及如何进行反向传播迭代。然后定义一个 `neuralNet` 结构体，其中定义一个受训神经网络所需的所有信息。稍后，将利用训练过的 `neuralNet` 值进行预测，这些定义如下面代码所示：

```
// neuralNet contains all of the information
// that defines a trained neural network.
type neuralNet struct {
    config neuralNetConfig
    wHidden *mat.Dense
    bHidden *mat.Dense
    wOut *mat.Dense
    bOut *mat.Dense
}

// neuralNetConfig defines our neural network
// architecture and learning parameters.
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
type neuralNetConfig struct {
    inputNeurons int
    outputNeurons int
    hiddenNeurons int
    numEpochs int
    learningRate float64
}
```

这里，wHidden 和 bHidden 分别是网络隐藏层的权重和偏差。wOut 和 bOut 分别是网络输出层的权重和偏差。请注意，这里使用 gonum.org/v1/gonum/mat 矩阵来计算所有的权重和偏差，使用相似矩阵来表示输入和输出。这样可以轻松地执行与反向传播相关的操作，并可以将训练推广到输入层、隐藏层和输出层中任意数量的节点。

接下来，定义一个函数，它基于 neuralNetConfig 的值初始化一个新的神经网络和基于输入矩阵（x）和输出矩阵（y）来训练 neuralNet 值的方法：

```
// NewNetwork initializes a new neural network.
func newNetwork(config neuralNetConfig) *neuralNet {
        return &neuralNet{config: config}
}

// Train trains a neural network using backpropagation.
func (nn *neuralNet) train(x, y *mat.Dense) error {

    // To be filled in...
}
```

在 train() 函数中，需要完成反向传播的方法，并将产生的训练权重和偏差放入接收器。首先，在 train() 方法中随机初始化权重和偏差，如下面代码所示：

```
// Initialize biases/weights.
randSource := rand.NewSource(time.Now().UnixNano())
randGen := rand.New(randSource)

wHiddenRaw := make([]float64,
nn.config.hiddenNeurons*nn.config.inputNeurons)
bHiddenRaw := make([]float64, nn.config.hiddenNeurons)
wOutRaw := make([]float64, nn.config.outputNeurons*nn.config.hiddenNeurons)
bOutRaw := make([]float64, nn.config.outputNeurons)

for _, param := range [][]float64{wHiddenRaw, bHiddenRaw, wOutRaw, bOutRaw}
{
    for i := range param {
        param[i] = randGen.Float64()
    }
}

wHidden := mat.NewDense(nn.config.inputNeurons, nn.config.hiddenNeurons,
wHiddenRaw)
bHidden := mat.NewDense(1, nn.config.hiddenNeurons, bHiddenRaw)
wOut := mat.NewDense(nn.config.hiddenNeurons, nn.config.outputNeurons,
wOutRaw)
bOut := mat.NewDense(1, nn.config.outputNeurons, bOutRaw)
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

然后，然后遍历每个阶段，完成网络的反向传播。这涉及计算输出的前馈阶段和应用权重、偏差变化的反向传播阶段。反向传播在附录中有更详细讨论，如有兴趣可以更深入研究。现在，主要是关注实现。阶段的循环结束看起来如下所示，反向传播过程的各个部分由注释表示。

```
// Define the output of the neural network.
output := mat.NewDense(0, 0, nil)

// Loop over the number of epochs utilizing
// backpropagation to train our model.
for i := 0; i < nn.config.numEpochs; i++ {

    // Complete the feed forward process.

    ...

    // Complete the backpropagation.

    ...

    // Adjust the parameters.

    ...
}
```

在这个循环的前馈阶段，输入通过节点网络向前传播。

```
// Complete the feed forward process.
hiddenLayerInput := mat.NewDense(0, 0, nil)
hiddenLayerInput.Mul(x, wHidden)
addBHidden := func(_, col int, v float64) float64 { return v +
bHidden.At(0, col) }
hiddenLayerInput.Apply(addBHidden, hiddenLayerInput)
hiddenLayerActivations := mat.NewDense(0, 0, nil)
applySigmoid := func(_, _ int, v float64) float64 { return sigmoid(v) }
hiddenLayerActivations.Apply(applySigmoid, hiddenLayerInput)

outputLayerInput := mat.NewDense(0, 0, nil)
outputLayerInput.Mul(hiddenLayerActivations, wOut)
addBOut := func(_, col int, v float64) float64 { return v + bOut.At(0, col)
}
outputLayerInput.Apply(addBOut, outputLayerInput)
output.Apply(applySigmoid, outputLayerInput)
```

然后，当有来自前馈过程的输出时，可以通过计算输出层和隐藏层的增量（或变化）用以反馈，如下面代码所示：

```
// Complete the backpropagation.
networkError := mat.NewDense(0, 0, nil)
networkError.Sub(y, output)

slopeOutputLayer := mat.NewDense(0, 0, nil)
applySigmoidPrime := func(_, _ int, v float64) float64 { return
sigmoidPrime(v) }
slopeOutputLayer.Apply(applySigmoidPrime, output)
```

仅供非商业用途或交流学习使用

```
slopeHiddenLayer := mat.NewDense(0, 0, nil)
slopeHiddenLayer.Apply(applySigmoidPrime, hiddenLayerActivations)

dOutput := mat.NewDense(0, 0, nil)
dOutput.MulElem(networkError, slopeOutputLayer)
errorAtHiddenLayer := mat.NewDense(0, 0, nil)
errorAtHiddenLayer.Mul(dOutput, wOut.T())

dHiddenLayer := mat.NewDense(0, 0, nil)
dHiddenLayer.MulElem(errorAtHiddenLayer, slopeHiddenLayer)
```

然后增量被用作更新网络的权重和偏差。**学习率**（learning rate）也被用于衡量这些变化，这样做有助于算法收敛。反向传播循环的最后一部分在这里实现：

```
// Adjust the parameters.
wOutAdj := mat.NewDense(0, 0, nil)
wOutAdj.Mul(hiddenLayerActivations.T(), dOutput)
wOutAdj.Scale(nn.config.learningRate, wOutAdj)
wOut.Add(wOut, wOutAdj)

bOutAdj, err := sumAlongAxis(0, dOutput)
if err != nil {
    return err
}
bOutAdj.Scale(nn.config.learningRate, bOutAdj)
bOut.Add(bOut, bOutAdj)

wHiddenAdj := mat.NewDense(0, 0, nil)
wHiddenAdj.Mul(x.T(), dHiddenLayer)
wHiddenAdj.Scale(nn.config.learningRate, wHiddenAdj)
wHidden.Add(wHidden, wHiddenAdj)

bHiddenAdj, err := sumAlongAxis(0, dHiddenLayer)
if err != nil {
    return err
}
bHiddenAdj.Scale(nn.config.learningRate, bHiddenAdj)
bHidden.Add(bHidden, bHiddenAdj)
```

请注意，这里利用了另一个辅助函数，使得在保持其他维度不变的情况下，沿着一个维度对矩阵求和。为了保证功能完整，在如下代码中实现 `sumAlongAxis()` 函数。

```
// sumAlongAxis sums a matrix along a
// particular dimension, preserving the
// other dimension.
func sumAlongAxis(axis int, m *mat.Dense) (*mat.Dense, error) {

    numRows, numCols := m.Dims()

    var output *mat.Dense

    switch axis {
    case 0:
        data := make([]float64, numCols)
        for i := 0; i < numCols; i++ {
            col := mat.Col(nil, i, m)
            data[i] = floats.Sum(col)
```

仅供非商业用途或交流学习使用

```
        }
        output = mat.NewDense(1, numCols, data)
    case 1:
        data := make([]float64, numRows)
        for i := 0; i < numRows; i++ {
            row := mat.Row(nil, i, m)
            data[i] = floats.Sum(row)
        }
        output = mat.NewDense(numRows, 1, data)
    default:
        return nil, errors.New("invalid axis, must be 0 or 1")
    }

    return output, nil
}
```

train() 方法最后要做的是将训练后的权重和偏差添加到接收器值中并返回。

```
// Define our trained neural network.
nn.wHidden = wHidden
nn.bHidden = bHidden
nn.wOut = wOut
nn.bOut = bOut

return nil
```

太棒了，这并没有那么复杂。用大约 100 行 Go 代码实现了神经网络的训练方法。为了检查这段代码运行状况，并了解权重和偏差大概是什么，让我们来使用一些简单的虚拟数据去训练神经网络。下一节会在网络中运行一个更真实的例子。但现在，先用一些虚拟数据进行一些自检，代码如下所示：

```
// Define our input attributes.
input := mat.NewDense(3, 4, []float64{
    1.0, 0.0, 1.0, 0.0,
    1.0, 0.0, 1.0, 1.0,
    0.0, 1.0, 0.0, 1.0,
})

// Define our labels.
labels := mat.NewDense(3, 1, []float64{1.0, 1.0, 0.0})

// Define our network architecture and
// learning parameters.
config := neuralNetConfig{
    inputNeurons: 4,
    outputNeurons: 1,
    hiddenNeurons: 3,
    numEpochs: 5000,
    learningRate: 0.3,
}

// Train the neural network.
network := newNetwork(config)
if err := network.train(input, labels); err != nil {
    log.Fatal(err)
}
```

仅供非商业用途或交流学习使用

```
// Output the weights that define our network!
f := mat.Formatted(network.wHidden, mat.Prefix(" "))
fmt.Printf("\nwHidden = % v\n\n", f)

f = mat.Formatted(network.bHidden, mat.Prefix(" "))
fmt.Printf("\nbHidden = % v\n\n", f)

f = mat.Formatted(network.wOut, mat.Prefix(" "))
fmt.Printf("\nwOut = % v\n\n", f)

f = mat.Formatted(network.bOut, mat.Prefix(" "))
fmt.Printf("\nbOut = % v\n\n", f)
```

编译和运行这个神经网络训练代码，得到如下权重和偏差的输出。

```
bOut$ go build
$ ./myprogram

wHidden = ⎡ 2.105009524680073 2.022752980359344 2.5200192466310547⎤
          ⎢ -2.033896626042324 -1.8652059633980662 -1.5861504822640748⎥
          ⎢ 1.821046795894909 2.436963623058306 1.717510016887383⎥
          ⎣-0.7400175664445425 -0.5800261988090052 -0.9277709997772002⎦


bHidden = [ 0.06850421921273529 -0.40252869229501825 -0.03392255287230178]


wOut = ⎡ 2.321901257946553⎤
       ⎢ 3.343178681830189⎥
       ⎣1.7581701319375231⎦


bOut = [-3.471613333924047]
```

这些权重和偏差能完全定义训练的神经网络，也就是说，wHidden 和 bHidden 允许将输入转换为网络输出层的输入。wOut 和 bOut 允许将这些值转换为神经网络的最终输出。

由于使用了随机数，你的程序返回结果可能和之前显示的有些不同，但是可以看到类似的取值范围。

8.3　使用简单的神经网络

现在有了一些看起来能工作的神经网络训练功能，接下来尝试在更真实的建模场景中使用这个功能。这里回到我们最喜爱的分类数据集合，即鸢尾花数据集（在第 5 章中采用的）。

如果你还记得，当使用这个数据集对鸢尾花进行分类时，尝试将其分为三类（setosa、virginica 或 versicolor）。由于神经网络是基于浮点类型的矩阵，需要将这

仅供非商业用途或交流学习使用

三个物种编码为数字列。一种方法是在每个物种的数据集中创建一个列。然后，根据相应行的测量值是否与该物种（1.0）或其他物种（0.0）相对应，将该列的值设置为 1.0 或 0.0。

我们也将标准化数据，原因与第 5 章中讨论的逻辑回归类似。这使得数据集看起来与之前有些不同，如下所示：

```
$ head train.csv
sepal_length,sepal_width,petal_length,petal_width,setosa,virginica,versicol
or
0.305555555556,0.583333333333,0.118644067797,0.0416666666667,1.0,0.0,0.0
0.25,0.291666666667,0.491525423729,0.541666666667,0.0,0.0,1.0
0.194444444444,0.5,0.0338983050847,0.0416666666667,1.0,0.0,0.0
0.833333333333,0.375,0.898305084746,0.708333333333,0.0,1.0,0.0
0.555555555556,0.208333333333,0.661016949153,0.583333333333,0.0,0.0,1.0
0.416666666667,0.25,0.508474576271,0.458333333333,0.0,0.0,1.0
0.666666666667,0.416666666667,0.677966101695,0.666666666667,0.0,0.0,1.0
0.416666666667,0.291666666667,0.491525423729,0.458333333333,0.0,0.0,1.0
0.5,0.416666666667,0.661016949153,0.708333333333,0.0,1.0,0.0
```

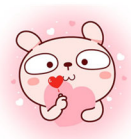
我们将使用 80/20 训练和测试集划分来评估模型。训练数据将保存在 `train.csv`，测试数据将保存在 `test.csv`，如下所示：

```
$ wc -l *.csv
  31 test.csv
 121 train.csv
 152 total
```

8.3.1 在实际数据上训练神经网络

为了在这个鸢尾花数据上训练神经网络，需要读入训练数据并创建两个矩阵。第一个矩阵将保存所有的属性（`inputs` 矩阵），第二个矩阵将保存所有的标签（`labels` 矩阵）。如下所示构建这些矩阵：

```
// Open the training dataset file.
f, err := os.Open("train.csv")
if err != nil {
    log.Fatal(err)
}
defer f.Close()

// Create a new CSV reader reading from the opened file.
reader := csv.NewReader(f)
reader.FieldsPerRecord = 7

// Read in all of the CSV records
rawCSVData, err := reader.ReadAll()
if err != nil {
    log.Fatal(err)
}

// inputsData and labelsData will hold all the
```



仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
// float values that will eventually be
// used to form our matrices.
inputsData := make([]float64, 4*len(rawCSVData))
labelsData := make([]float64, 3*len(rawCSVData))

// inputsIndex will track the current index of
// inputs matrix values.
var inputsIndex int
var labelsIndex int

// Sequentially move the rows into a slice of floats.
for idx, record := range rawCSVData {

    // Skip the header row.
    if idx == 0 {
        continue
    }

    // Loop over the float columns.
    for i, val := range record {

        // Convert the value to a float.
        parsedVal, err := strconv.ParseFloat(val, 64)
        if err != nil {
            log.Fatal(err)
        }

        // Add to the labelsData if relevant.
        if i == 4 || i == 5 || i == 6 {
            labelsData[labelsIndex] = parsedVal
            labelsIndex++
            continue
        }

        // Add the float value to the slice of floats.
        inputsData[inputsIndex] = parsedVal
        inputsIndex++
    }
}

// Form the matrices.
inputs := mat.NewDense(len(rawCSVData), 4, inputsData)
labels := mat.NewDense(len(rawCSVData), 3, labelsData)
```

然后，初始化和训练神经网络，就像处理虚拟数据一样。唯一的区别是将使用对应于三个输出类的三个输出神经元。训练过程如下：

```
// Define our network architecture and
// learning parameters.
config := neuralNetConfig{
    inputNeurons:  4,
    outputNeurons: 3,
    hiddenNeurons: 3,
    numEpochs:     5000,
    learningRate:  0.3,
}

// Train the neural network.
```

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用


```
network := newNetwork(config)
if err := network.train(inputs, labels); err != nil {
    log.Fatal(err)
}
```

8.3.2 评估神经网络

为了预测训练的神经网络，将为neuralNet（神经网络）类型创建一个predict()方法。这个predict方法将接收新的输入，并通过网络完成并向前馈送新输入以产生预测输出，如下所示：

```
// predict makes a prediction based on a trained
// neural network.
func (nn *neuralNet) predict(x *mat.Dense) (*mat.Dense, error) {

    // Check to make sure that our neuralNet value
    // represents a trained model.
    if nn.wHidden == nil || nn.wOut == nil || nn.bHidden == nil || nn.bOut
== nil {
        return nil, errors.New("the supplied neural net weights and biases
are empty")
    }

    // Define the output of the neural network.
    output := mat.NewDense(0, 0, nil)
github.com/tensorflow/tensorflow/tensorflow/go
    // Complete the feed forward process.
    hiddenLayerInput := mat.NewDense(0, 0, nil)
    hiddenLayerInput.Mul(x, nn.wHidden)
    addBHidden := func(_, col int, v float64) float64 { return v +
nn.bHidden.At(0, col) }
    hiddenLayerInput.Apply(addBHidden, hiddenLayerInput)

    hiddenLayerActivations := mat.NewDense(0, 0, nil)
    applySigmoid := func(_, _ int, v float64) float64 { return sigmoid(v) }
    hiddenLayerActivations.Apply(applySigmoid, hiddenLayerInput)

    outputLayerInput := mat.NewDense(0, 0, nil)
    outputLayerInput.Mul(hiddenLayerActivations, nn.wOut)
    addBOut := func(_, col int, v float64) float64 { return v +
nn.bOut.At(0, col) }
    outputLayerInput.Apply(addBOut, outputLayerInput)
    output.Apply(applySigmoid, outputLayerInput)

    return output, nil
}
```

然后可以在测试数据上使用这个predict()方法来评估训练好的模型。具体来说，将test.csv中的数据读入两个新的矩阵testInputs和testLabels，类似于在train.csv中读取内容（因此，在此不会详述）。可以将testInputs提供给predict()方法以获得预测，然后可以比较预测和testLabels以计算评估指标。此处将计算模型的准确度。

仅供非商业用途或交流学习使用

在计算准确度时，需要关注的一个细节是根据模型确定单个输出预测。网络实际上为每种鸢尾花种类产生 0.0 到 1.0 的输出。这里将选择最高的数值作为所预测的物种。

以下代码显示了模型的准确度计算：

```
// Make the predictions using the trained model.
predictions, err := network.predict(testInputs)
if err != nil {
    log.Fatal(err)
}

// Calculate the accuracy of our model.
var truePosNeg int
numPreds, _ := predictions.Dims()
for i := 0; i < numPreds; i++ {

    // Get the label.
    labelRow := mat.Row(nil, i, testLabels)
    var species int
    for idx, label := range labelRow {
        if label == 1.0 {
            species = idx
            break
        }
    }

    // Accumulate the true positive/negative count.
    if predictions.At(i, species) == floats.Max(mat.Row(nil, i,
predictions)) {
        truePosNeg++
    }
}

// Calculate the accuracy (subset accuracy).
accuracy := float64(truePosNeg) / float64(numPreds)

// Output the Accuracy value to standard out.
fmt.Printf("\nAccuracy = %0.2f\n\n", accuracy)
```

编译和运行评估产生类似于以下内容的结果：

```
$ go build
$ ./myprogram

Accuracy = 0.97
```

这个结果看起来非常好 。用从零开始的神经网络预测鸢尾花物种的准确率为 97%。正如前面所提到的，对于大多数分类任务，这些单隐藏层神经网络非常强大，并且可以看到底层原理并不是那么复杂。

8.4　引入深度学习

尽管简单的神经网络（如前一节中所用的神经网络）对于许多场景来说非常强大，

仅供非商业用途或交流学习使用

但近年来，各种行业已经将深度神经网络架构应用于各种类型的数据。这些更复杂的架构已被用来击败棋盘 / 视频游戏的冠军，驱动汽车，生成艺术作品，转换图像等。看起来可以在这些模型上输出任何东西，它们会做一些有趣的事情，但它们似乎特别适合于计算机视觉、语音识别、文本推理以及其他非常复杂和难以定义的任务。

这里将介绍深度学习，并在 Go 中运行深度学习模型。然而，深度学习模型的应用和多样性是巨大而且每日剧增的。

关于这个主题有很多书籍和教程，所以如果这个简短的介绍引起了你的兴趣，建议查看以下资源之一：

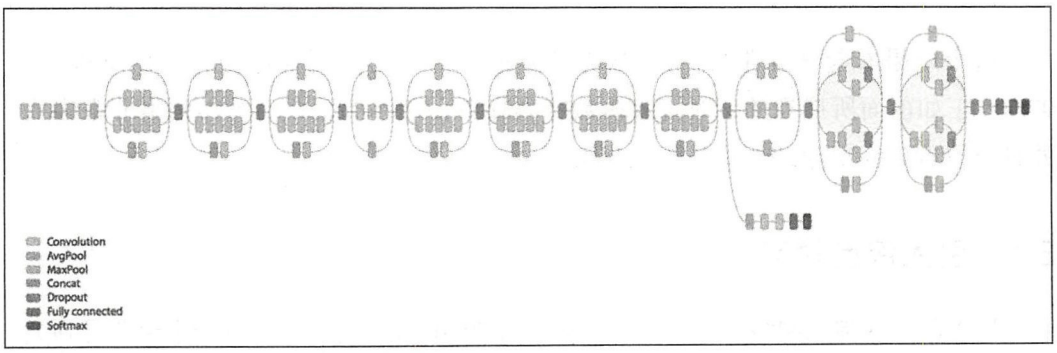
- 用 TensorFlow 实践深度学习：https://www.packtpub.com/big-data-and-business-intelligence/hands-deep-learning-tensorflow
- 深度学习示例：https://www.packtpub.com/big-data-and-business-intelligence/deep-learning-example
- 计算机视觉的深度学习：https://www.packtpub.com/big-data-and-business-intelligence/deep-learning-computer-vision

这些资源可能不会全部提及或支持 Go，但正如下面的示例中所看到的，Go 完全可以应用这些技术并与 TensorFlow 或 H2O 等接口连接。

8.4.1 什么是深度学习模型

深度学习的深度是指连续组合和利用各种神经网络结构，形成庞大而复杂的架构。这些庞大而复杂的架构通常需要大量的数据进行训练，所产生的结构很难解释。

为了举例说明现代深度学习模型的规模和复杂性，请以 Google 的 LeNet（http://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Szegedy_Going_Deeper_With_2015_CVPR_paper.pdf）为例。这个模型可用于物体识别，此处会有描述（图片提供自 https://adeshpande3.github.io/adeshpande3.github.io/The-9-Deep-Learning-Papers-You-Need-To-Know-About.html）：



仅供非商业用途或交流学习使用

这种结构本身就很难理解。然而，当你意识到图中的每个块本身可能是一个复杂的神经网络（如图所示）时，它会更加令人印象深刻。

通常，深度学习模型通过链接在一起的方式构建，并且在一些情况下会重新链接在一起，构建一个或多个以下结构：

- 一个全连接神经网络架构：这个构建块可能与在本章之前为鸢尾花检测所创建的东西类似。其中包含的神经网络节点以全连接层方式排列。也就是说，这些层从一个流向另一个，所有节点从一个层连接到另一个层。
- **卷积神经网络**（ConvNet 或 CNN）：至少实现了一个**卷积层**（convolutional layer）的神经网络。卷积层包含由一组权重（也称为卷积核或过滤器）参数化的神经元，这些神经元仅部分地连接到输入数据。大多数卷积神经网络包含这些卷积层的分层组合，允许卷积神经网络响应输入数据中的低级特征。
- **递归神经网络**和 / 或**长短期记忆**（LSTM）**细胞**：这些组件试图将诸如记忆之类的因素考虑进模型中。递归神经网络使其输出依赖于一系列输入，而不是假设每个输入都是独立的。LSTM 细胞与递归神经网络有关，但也试图考虑更长期的记忆。

虽然本章将利用图像数据来说明深度学习，但递归神经网络和 LSTM 的深度学习模型特别适用于解决基于文本和其他 NLP 问题。例如，内置于这些网络中的内存使他们能够理解句子中下一个可能出现的单词。如果你有兴趣运行基于文本数据的深度学习模型，建议查看使用 `github.com/chewxy/gorgonia` 软件包的此示例：https://github.com/chewxy/gorgonia/tree/master/examples/charRNN。

深度学习模式非常强大！尤其适用于计算机视觉等任务。但是，还应该记住，这些神经网络组件的复杂组合也非常难以解释。也就是说，确定模型为什么会做出某种预测几乎是不可能的。当需要维护某些行业和辖区的合规性时，这可能会成为问题，并且还可能会妨碍应用程序的调试或维护。这就是说，为了提高深度学习模型的可解释性，还有很多工作要做。这些工作中值得注意的是 LIME 项目：https://github.com/marcotcr/lime。

8.4.2　基于 Go 语言的深度学习

当想要构建或利用 Go 的深度学习模型时，有多种选择。与深度学习本身一样，这是一个不断变化的生态。但是，在 Go 中构建、训练和利用深度学习模型的选项

仅供非商业用途或交流学习使用

大致如下：

- **使用 Go 软件包**：Go 软件包允许你使用 Go 作为主要接口来构建和训练深度学习模型。github.com/chewxy/gorgonia.github.com/chewxy/gorgonia 最大的特点是将 Go 当做一等公民，并以 Go 来编写，即使它确实大量使用 cgo 同数值类库进行交互。
- **对于非 Go DL 框架使用 API 或者 Go 客户端**：可以从 Go 中与当前流行的深度学习服务和框架进行交互，这些服务和框架包括 TensorFlow、MachineBox、H2O 以及各种云服务商或第三方 API 产品（如 IBM Watson）。TensorFlow 和 Machine Box 实际上有 Go 绑定或 SDK，并且还在不断改进中。对于其他服务，你可能需要通过 REST 进行交互，甚至使用 exec 调用二进制文件。
- **使用 cgo**：当然，Go 可以与 C/C++ 库进行交互和集成用以深度学习，其中包括 TensorFlow 库和来自英特尔的各种库。但是，这是一条艰难的道路，只有在绝对必要时才推荐。

由于 TensorFlow 是迄今为止最流行的深度学习框架，接下来将简要探讨此处列出的第二类。但是，由于 Go 绑定的 Tensorflow 正在处于积极开发之中，目前一些功能还比较粗糙。TensorFlow 团队建议，如果要在 Go 中使用 TensorFlow 模型，首先使用 Python 训练并导出该模型。这个预先训练好的模型可以在 Go 中使用，这将在下一节中演示。

在社区中有很多非常努力的成员，使得 Go 在 TensorFlow 中越来越重要。因此，TensorFlow 绑定的粗糙的接口很可能会在未来一年内更加完善。为了在这方面保持更新，请确保并加入 Gophers Slack 上的 # data-science 频道（https://invite.slack.golangbridge.org/）。这是一个经常讨论的社区，并且会成为提问和参与的好地方。

尽管将在这里探索一个快速的 TensorFlow 示例，但仍强烈建议查看 github.com/chewxy/gorgonia，特别是如果正考虑在 Go 中执行更多定制化的和广泛的建模。这个软件包功能超强，并已经在实际生产中使用。

8.4.2.1 设置 TensorFlow 与 Go 一起使用

TensorFlow 团队提供了一些很好的文档来安装 TensorFlow 并准备好与 Go 一起使用。这些文档可以在 https://www.tensorflow.org/install/install_go 找到。有几个初步步骤，一旦安装了 TensorFlow C 库，可以获得以下 Go 包：

```
$ go get github.com/tensorflow/tensorflow/tensorflow/go
```

如果能够分毫无差地获得 github.com/tensorflow/tensorflow/tensorflow/go，那么一切都应该顺利，可以执行以下测试来确保已准备好使用 TensorFlow：

```
$ go test github.com/tensorflow/tensorflow/tensorflow/go
ok github.com/tensorflow/tensorflow/tensorflow/go 0.045s
```

8.4.2.2　检索并调用预训练的 TensorFlow 模型

这里要使用的模型是一个 Google 模型，用于识别图像中的物体，称为“先觉”。该模型可用如下方法获取：

```
$ mkdir model
$ cd model
$ wget
https://storage.googleapis.com/download.tensorflow.org/models/inception5h.z
ip
--2017-09-09 18:29:03--
https://storage.googleapis.com/download.tensorflow.org/models/inception5h.z
ip
Resolving storage.googleapis.com (storage.googleapis.com)... 172.217.6.112,
2607:f8b0:4009:812::2010
Connecting to storage.googleapis.com
(storage.googleapis.com)|172.217.6.112|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 49937555 (48M) [application/zip]
Saving to: 'inception5h.zip'

inception5h.zip
100%[=========================================================================
=================================================>] 47.62M 19.0MB/s in
2.5s

2017-09-09 18:29:06 (19.0 MB/s) - 'inception5h.zip' saved [49937555/49937555]

$ unzip inception5h.zip
Archive: inception5h.zip
inflating: imagenet_comp_graph_label_strings.txt
inflating: tensorflow_inception_graph.pb
inflating: LICENSE
```

解压打包的模型后，你应该看到一个 *.pb 文件。这是一个代表模型冻结状态的 protobuf 文件。回想一下之前介绍的简单神经网络，网络完全由一系列的权重和偏差定义。尽管更复杂，但可以用类似的方式定义此模型，并将这些定义存储在此 protobuf 文件中。

要调用此模型，将使用 TensorFlow Go 绑定文档中的一些示例代码——https://godoc.org/github.com/tensorflow/tensorflow/tensorflow/go。此代码加载模型并使用该模型检测和标记 *.jpg 图像的内容。

由于代码包含在 TensorFlow 文档中，将省略细节并突出显示几个片段。要加载模型，可以执行以下操作：

仅供非商业用途或交流学习使用

```
// Load the serialized GraphDef from a file.
modelfile, labelsfile, err := modelFiles(*modeldir)
if err != nil {
    log.Fatal(err)
}
model, err := ioutil.ReadFile(modelfile)
if err != nil {
    log.Fatal(err)
}
```

然后加载深度学习模型的图形定义，并用图形创建一个新的 TensorFlow 会话，如下面的代码所示：

```
// Construct an in-memory graph from the serialized form.
graph := tf.NewGraph()
if err := graph.Import(model, ""); err != nil {
    log.Fatal(err)
}

// Create a session for inference over graph.
session, err := tf.NewSession(graph, nil)
if err != nil {
    log.Fatal(err)
}
defer session.Close()
```

最后，可以使用该模型进行如下推理：

```
// Run inference on *imageFile.
// For multiple images, session.Run() can be called in a loop (and
// concurrently). Alternatively, images can be batched since the model
// accepts batches of image data as input.
tensor, err := makeTensorFromImage(*imagefile)
if err != nil {
    log.Fatal(err)
}
output, err := session.Run(
    map[tf.Output]*tf.Tensor{
        graph.Operation("input").Output(0): tensor,
    },
    []tf.Output{
        graph.Operation("output").Output(0),
    },
    nil)
if err != nil {
    log.Fatal(err)
}

// output[0].Value() is a vector containing probabilities of
// labels for each image in the "batch". The batch size was 1.
// Find the most probable label index.
probabilities := output[0].Value().([][]float32)[0]
printBestLabel(probabilities, labelsfile)
```

8.4.2.3 使用 Go 的 TensorFlow 进行对象检测

在 TensorFlow GoDocs 中指定的用于对象检测的 Go 程序可以按如下方式调用：

仅供非商业用途或交流学习使用

```
$ ./myprogram -dir=<path/to/the/model/dir> -image=<path/to/a/jpg/image>
```

当程序被调用时，它将利用预训练的和加载的模型来推断指定图像的内容。然后它会输出该图像最可能的内容及其计算出的概率。

为了说明这一点，将会尝试在下面的飞机图像上执行对象检测，图像保存为 airplane.jpg：

从 Go 运行 TensorFlow 模型会得到以下结果：

```
$ go build
$ ./myprogram -dir=model -image=airplane.jpg
2017-09-09 20:17:30.655757: W
tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use SSE4.1 instructions, but these are available on your
machine and could speed up CPU computations.
2017-09-09 20:17:30.655807: W
tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use SSE4.2 instructions, but these are available on your
machine and could speed up CPU computations.
2017-09-09 20:17:30.655814: W
tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use AVX instructions, but these are available on your
machine and could speed up CPU computations.
2017-09-09 20:17:30.655818: W
tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use AVX2 instructions, but these are available on your
machine and could speed up CPU computations.
2017-09-09 20:17:30.655822: W
tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use FMA instructions, but these are available on your
machine and could speed up CPU computations.
BEST MATCH: (86% likely) airliner
```

在提出了一些关于加速 CPU 计算的建议之后，得到了一个结果：airliner。哇！这很酷。刚刚从 Go 程序中利用 TensorFlow 执行了对象识别！

接下来尝试另一个。这一次，将使用 pug.jpg，如下所示：

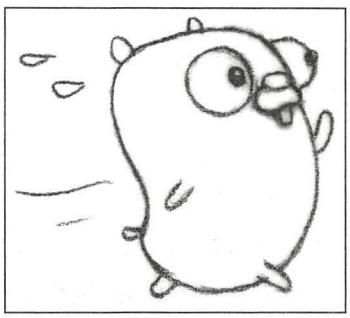
仅供非商业用途或交流学习使用

再次对此图像运行程序，得到如下内容：

```
$ ./myprogram -dir=model -image=pug.jpg
2017-09-09 20:20:32.323855: W
tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use SSE4.1 instructions, but these are available on your
machine and could speed up CPU computations.
2017-09-09 20:20:32.323896: W
tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use SSE4.2 instructions, but these are available on your
machine and could speed up CPU computations.
2017-09-09 20:20:32.323902: W
tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use AVX instructions, but these are available on your
machine and could speed up CPU computations.
2017-09-09 20:20:32.323906: W
tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use AVX2 instructions, but these are available on your
machine and could speed up CPU computations.
2017-09-09 20:20:32.323911: W
tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use FMA instructions, but these are available on your
machine and could speed up CPU computations.
BEST MATCH: (84% likely) pug
```

成功了！模型不仅检测到图片中有一只狗，它正确地确定图片中有一只哈巴狗。

来再尝试一次。由于这是一本 Go 书，我们无法抗拒尝试 `gopher.jpg`，看起来像下面这样（非常感谢 Go gopher 背后的艺术家 Renee French）：



运行模型会得出以下结果：

仅供非商业用途或交流学习使用

```
$ ./myprogram -dir=model -image=gopher.jpg
2017-09-09 20:25:57.967753: W
tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use SSE4.1 instructions, but these are available on your
machine and could speed up CPU computations.
2017-09-09 20:25:57.967801: W
tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use SSE4.2 instructions, but these are available on your
machine and could speed up CPU computations.
2017-09-09 20:25:57.967808: W
tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use AVX instructions, but these are available on your
machine and could speed up CPU computations.
2017-09-09 20:25:57.967812: W
tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use AVX2 instructions, but these are available on your
machine and could speed up CPU computations.
2017-09-09 20:25:57.967817: W
tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library
wasn't compiled to use FMA instructions, but these are available on your
machine and could speed up CPU computations.
BEST MATCH: (12% likely) safety pin
```

可能这个模型无法适用于所有的场景。看起来需要重构这个模型，以便能够识别 Go gopher。更具体地说，可能应该在训练数据集中添加一堆 Go gopher，因为 Go gopher 绝对不是安全别针！

8.5　参考书目

基本神经网络：

- 快速介绍神经网络：https://ujjwalkarn.me/2016/08/09/quick-intro-neural-networks/
- 深度学习和神经网络的友好介绍：https://youtu.be/BR9h47Jtqyw
- `github.com/tleyden/neurgo` 文档：https://godoc.org/github.com/tleyden/neurgo
- `github.com/fxsjy/gonn/gonn` 文档：https://godoc.org/github.com/fxsjy/gonn/gonn

更详细的深度学习资源：

- 用 TensorFlow 实践深度学习：https://www.packtpub.com/big-data-and-business-intelligence/hands-deep-learning-tensorflow
- 深度学习示例：https://www.packtpub.com/big-data-and-business-intelligence/deep-learning-example
- 深度学习计算机视觉：https://www.packtpub.com/big-data-and-business-intelligence/deep-learning-computer-vision

仅供非商业用途或交流学习使用

Go 的深度学习：

- TensorFlow Go 绑定文档：https://godoc.org/github.com/tensorflow/tensorflow/tensorflow/go
- `github.com/chewxy/gorgonia` 文档：https://godoc.org/github.com/chewxy/gorgonian
- MachineBox Go SDK 文档：https://godoc.org/github.com/machinebox/sdk-go
- 使用 `cgo` 调用预训练模型示例：https://github.com/gopherdata/dlinfer

8.6 小结

恭喜！到此已经完成了从使用 Go 解析数据到使用 Go 调用深度学习模型。现在已经介绍完神经网络的基础知识，并且可以在 Go 程序中实现和使用它们。在下一章中，将讨论如何从笔记本电脑中获取这些模型和应用程序，并在数据流水线中以生产规模运行它们。

仅供非商业用途或交流学习使用

第 9 章

部署、分布分析和模型

现在已经利用 Go 实现了很多类型的模型，包括回归、分类、集群等。同时还学习了关于开发机器学习模型的一些流程。这些模型成功预测了疾病进展、花卉物种和图像中的物体。但是，仍然没有谈到机器学习难题中的一个重要组成部分：部署、维护和拓展。

如果模型仅仅停留在笔记本上，那么在公司内部不会带来任何好处，也不会有任何价值。不仅需要知道如何使用机器学习工作流程，并且将其集成到公司已经部署的系统中，还需要知道随着时间推移，如何拓展、更新和维护这些工作流程。

事实上，机器学习的工作流程很自然地就是多阶段的工作流程，这可能会使部署和维护成为一项挑战。需要训练、测试和利用模型，并且在某些情况下，需要对数据进行预处理和 / 或后处理。可能还需要将某些模型链接在一起。如何部署和连接所有这些阶段，同时保持应用程序的简单性和完整性呢？例如，如何随时间更新训练数据集，同时仍然知道哪些训练数据集生成了哪些模型，以及哪些模型生成了哪些数据？随着对这些预测的需求不断上升和下降，如何轻松地伸缩我们的预测？最后，如何将机器学习的工作流程与基础设施中其他应用，或者基础设施本身部分（数据库、队列等）整合在一起？

本章将解决所有这些问题。事实证明，使用 Go 语言本身和用 Go 语言编写的基础架构为部署和管理机器学习工作流程提供了非常好的平台。这里将从上至下完全采用基于 Go 的方法，并说明每个部分如何帮助我们大规模开展数据科学研究。

9.1 在远程机器上可靠地运行模型

无论公司是使用企业内部基础架构还是云架构，都需要将机器学习模型运行在

仅供非商业用途或交流学习使用

除了笔记本以外的某个地方。这些模型可能需要准备为欺诈预测提供服务，或者需要实时处理用户上传的图像。仅仅让模型工作在笔记本电脑上是无法成功提供这些信息的。

但是，当使用笔记本电脑获得数据处理和机器学习应用时，应该确保以下内容：

1. 不应该为了部署和拓展而将应用程序复杂化。应该保持应用简单，这样有助于维护和确保完整性。
2. 应该确保应用程序的行为与其在本地开发计算机上的行为一致。

如果部署后机器学习流程的表现和行为与开发阶段不同，那么将不能产生预期的数值。应该理解模型如何在本地工作，并且假设同生产环境中行为一致。如果在部署阶段增加很多不必要的复杂逻辑，会使得机器学习变得越来越难以控制。

使用 Docker(https://www.docker.com/) 是使部署简单、易携带和可重用的一种方法，这里将利用它来部署机器学习应用程序。

Docker 本身是使用 Go 编写的，这使得它成为在机器学习部署栈中第一个基于 Go 的架构工具。

9.1.1 Docker 和 Docker 术语简介

Docker 和整个容器生态系统都有自己一套术语，这可能会让人困惑，尤其是对于那些有虚拟机经验的人来说。在继续之前，先巩固以下术语：

- **Docker 镜像**是一个数据层的集合，它们定义了一个文件系统、库、环境变量等，这些内容对运行于容器内的应用是可见的。将镜像想象为一个软件包，其中包含了应用程序和其他相关库或软件包，以及其他应用程序运行所需的环境部分。一个 Docker 镜像并不包括完整的操作系统。
- **Dockerfile** 是一个定义 Docker 镜像各层的文件。
- **Docker engine** 可以帮助构建、管理和运行 Docker 镜像。
- 为应用构建 Docker 镜像的过程通常称为 **Docker 化**应用。
- **容器**或**软件容器**是 Docker 镜像的运行实例，本质上来讲，这个运行的容器包括 Docker 镜像的所有层，加上允许应用程序运行、输入 / 输出数据等的读 / 写层。
- **Docker registry** 是保存 Docker 镜像的地方。这个 registry 可以是本地的，也可以是在远端机器上运行。它还可以是托管的 registry 服务，例如 Docker Hub 或 Quay、Amazon Web Service（AWS）、Amazon EC2 Container Registry（ECR）等。

仅供非商业用途或交流学习使用

注意：Docker 镜像与虚拟机不同，Docker 镜像包括应用程序、文件系统以及各种库和软件包，但实际上它并不包含客户操作系统。而且，它运行时不占用主机一定数量的内存、磁盘和 CPU。Docker 容器共享 Docker engine 运行时底层内核的资源。

希望所有这些术语在本章后面的示例中得到巩固。就像本书的其他主题一样，可以对 Docker 以及软件容器更深入地了解。本章将引入更多 Docker 资源的链接。

在下面的示例中，将假设你已经能够在本地构建和运行 Docker 镜像。可以按照如下说明安装 Docker：https://www.docker.com/community-edition#/download。

9.1.2　Docker 化机器学习的应用

本章中部署和扩展的机器学习工作流程，将成为第 4 章中开发的预测糖尿病进展的线性回归工作流程。在部署中，会考虑三种不同的工作流程：

- 训练和导出一个单一回归模型（用身体质量指数为疾病进展建模）
- 训练和导出一个多元回归模型（用身体质量指数和血液测量 LTG 为疾病进展建模）
- 基于训练模型和输入属性对疾病进展的一个推断

在本章后面，将会知道为什么要将工作流程分割成这些部分。但是现在，主要精力是放在 Docker 化的工作流程。（也就是构建可以运行工作流程的 Docker 镜像）。

9.1.2.1　Docker 化模型的训练和导出

这里将使用与第 4 章相同的代码进行模型训练。但是，将对代码进行一些调整，使其更加便于用户使用，并能与工作流程中其他部分进行交互。这里不会考虑复杂的实际建模代码。相反，这些针对的是任何可能打算广泛使用的应用程序。

首先，将添加一些命令行标志到应用程序中，以便可以指定训练数据集所在的输入目录，以及指定用于导出持久化模型的一个输出目录。可以按如下方式实现这些命令行参数：

```
// Declare the input and output directory flags.
inDirPtr := flag.String("inDir", "", "The directory containing the training
data")
outDirPtr := flag.String("outDir", "", "The output directory")

// Parse the command line flags.
flag.Parse()
```

然后创建几个结构体类型，用以导出模型的系数和截距到 JSON 文件中。这个导出的 JSON 文件实际上是训练模型的一个持久化版本，因为系数和截距完全参数

仅供非商业用途或交流学习使用

化了模型。这些就结构体如下所示：

```
// ModelInfo includes the information about the
// model that is output from the training.
type ModelInfo struct {
    Intercept float64 `json:"intercept"`
    Coefficients []CoefficientInfo `json:"coefficients"`
}

// CoefficientInfo include information about a
// particular model coefficient.
type CoefficientInfo struct {
    Name string `json:"name"`
    Coefficient float64 `json:"coefficient"`
}
```

除此之外，训练的代码同第4章是一样的。这里仍然会使用github.com/sajari/regression来训练模型。然后将模型导出为JSON文件。单个回归模型的训练和导出包含在下面代码中：

```
// Train/fit the single regression model
// with github.com/sajari/regression
// like in Chapter 5, Regression.

// Fill in the model information.
modelInfo := ModelInfo{
    Intercept: r.Coeff(0),
    Coefficients: []CoefficientInfo{
        CoefficientInfo{
            Name: "bmi",
            Coefficient: r.Coeff(1),
        },
    },
}

// Marshal the model information.
outputData, err := json.MarshalIndent(modelInfo, "", " ")
if err != nil {
    log.Fatal(err)
}

// Save the marshalled output to a file, with
// certain permissions (http://permissions-calculator.org/decode/0644/).
if err := ioutil.WriteFile(filepath.Join(*outDirPtr, "model.json"),
outputData, 0644); err != nil {e model to the JSON fil
    log.Fatal(err)
}
```

然后，对于多元回归模型，该过程如下所示：

```
// Train/fit the multiple regression model
// with github.com/sajari/regression
// like in Chapter 5, Regression.

// Fill in the model information.
modelInfo := ModelInfo{
```

仅供非商业用途或交流学习使用

```
        Intercept: r.Coeff(0),
        Coefficients: []CoefficientInfo{
            CoefficientInfo{
                Name:        "bmi",
                Coefficient: r.Coeff(1),
            },
            CoefficientInfo{
                Name:        "ltg",
                Coefficient: r.Coeff(2),
            },
        },
    }

    // Marshal the model information.
    outputData, err := json.MarshalIndent(modelInfo, "", "    ")
    if err != nil {
        log.Fatal(err)
    }

    // Save the marshalled output to a file.
    if err := ioutil.WriteFile(filepath.Join(*outDirPtr, "model.json"),
    outputData, 0644); err != nil {
        log.Fatal(err)
    }
```

为了将训练过程 Docker 化，需要为每个一元回归训练程序和多元回归训练程序创建一个 Dockerfile。实际上，这些程序都可以用基本相同的 Dockerfile。这个 Dockerfile 和程序位于同一目录，如下所示：

```
FROM alpine
ADD goregtrain /
```

非常简单，对吧？下面来探讨这两行的功能。还记得 Docker 镜像是指定应用程序运行环境的一系列层吗？这个 Dockerfile 使用两个 Dockerfile 命令 `FROM` 和 `ADD` 构建了这些层。

`FROM alpine` 指定了 Docker 镜像的文件系统、应用程序和库是基于 Docker Hub 官方的 Alpine Linux Docker 镜像。使用 `alpine` 作为基础镜像的原因是它是一个非常小的 Docker 镜像（使得它非常便携），并且他它包含一些 Linux shell 的细节。

`ADD goregtrain/` 指定要将一个 `goregtrain` 文件添加到 Docker 镜像的 / 目录。这个 `goregtrain` 文件实际上是从 Go 代码中构建的 Go 二进制文件。因此，Dockerfile 中所有内容是说想要在 Alpine Linux 中运行我们的二进制 Go 文件。

除非使用 cgo 并依赖一堆外部的 C 库，否则应该在构建 Docker 镜像之前构建 Go 二进制文件，并将静态链接的二进制 Go 文件拷贝到 Docker 镜像中。这将加快 Docker 的构建速度，使 Docker 镜像非常小，这意味着可以轻松地将其移植到任何系统中，并以超快的速度启动它们。

现在，需要在构建 Docker 镜像之前构建二进制 Go 文件，并将二进制 Go 文件拷贝到镜像中。为此，使用如下所示的 Makefile：

```
all: compile docker push clean

compile:
        GOOS=linux GOARCH=amd64 CGO_ENABLED=0 go build -o goregtrain

docker:
        sudo docker build --force-rm=true -t dwhitena/goregtrain:single .

push:
        sudo docker push dwhitena/goregtrain:single

clean:
        rm goregtrain
```

如你所见：

- make compile 为目标架构编译二进制 Go 文件，并命名为 goregtrain。
- make docker 基于 Dockerfile 使用 Docker engine 构建镜像，并标记该镜像为 dwhitena/goregtrain:single。
- 标签中的 dwhitena 部分指明了存储镜像的 Docker Hub 用户名（本例中是 dwhitena），goregtrain 指明了镜像的名字，:single 指明了镜像标记的版本。
- 一旦镜像构建，make push 将推送最近构建的 Docker 镜像到 registry 中。在本例中，它将推送到 Docker Hub 的 dwhitena 用户名下（当然，可以推送到任何公开或私有的 registry 中）。
- 最后，make clean 将清理二进制文件。

如前所述，这个 Dockerfile 和 Makefile 对于单一回归和多元回归模型来说都是一样的。但是，将使用不同的 Docker 镜像标签来区分这两个模型。对于单一回归模型将使用 dwhitena/goregtrain:single，对于多元回归模型将使用 dwhitena/goregtrain:multi。

在这些例子和本章剩余部分，可以使用 Docker Hub 上 dwhitena 中公开的 Docker 镜像进行本地操作，这无须修改此处打印的示例。请注意，无法向 Docker Hub 上的 dwhitena 创建和推送镜像，因为你不是 dwhitena。或者，可以用你自己的 Docker Hub 用户名替代示例中任何出现 dwhitena 的地方。这将使你可以构建、推送和利用你自己的镜像。

为了在单一或多元回归模型构建 Docker 镜像之后构建、推送和清理，可以运行 make，如下面代码所示，它本身是使用 Go 编写的。

```
$ make
GOOS=linux GOARCH=amd64 CGO_ENABLED=0 go build -o goregtrain
sudo docker build --force-rm=true -t dwhitena/goregtrain:multi
[sudo] password for dwhitena:
Sending build context to Docker daemon 2.449MB
Step 1/2 : FROM alpine
 ---> 7328f6f8b418
Step 2/2 : ADD goregtrain /
 ---> 9c13594ad7de
Removing intermediate container 6e717232c6d1
Successfully built 9c13594ad7de
Successfully tagged dwhitena/goregtrain:multi
sudo docker push dwhitena/goregtrain:multi
The push refers to a repository [docker.io/dwhitena/goregtrain]
375642156c06: Pushed
5bef08742407: Layer already exists
multi: digest:
sha256:f35277a46ed840922a0d7e94c3c57632fc947f6ab004deed66d3eb80a331e40f
size: 738
rm goregtrain
```

从前面的输出可以看到，Docker engine 构建了镜像的两个层，标记镜像，然后推送镜像到 Docker Hub 中。现在可以通过以下方式在本地 registry 中看到 Docker 镜像。

```
$ sudo docker images | grep "goregtrain"
dwhitena/goregtrain multi 9c13594ad7de About a minute ago 6.41MB
```

同样可以在 Docker Hub 中看到它（https://hub.docker.com/r/dwhitena/goregtrain/），如下所示：

很快将看到如何运行和利用这个 Docker 镜像，但是首先构建另一个 Docker 镜像，用于基于 JSON- 持久化的模型来生成预测结果。

9.1.2.2　Docker 化模型预测

与模型的训练一样，我们将利用命令行参数指定预测程序所使用的输入目录和

输出目录。这次将有两个输入目录，一个用于持久化模型，另一个包含将要预测的属性。相应的程序如下所示：

1. 从模型输入目录读入模型。
2. 在属性输入目录中浏览文件。
3. 对于属性输入目录中的每个文件（包含没有相应疾病进展预测的属性），利用加载的模型预测疾病进展。
4. 将疾病进展输出到命令行参数指定的目录。

按如下思路思考该流程，我们基于历史数据训练了用于预测疾病进展的模型，希望医生或诊所以某种方式为新患者使用该预测。它们提供了患者属性（**身体质量指数**（BMI），或者身体质量指数和**长期增长**（LTG）），我们根据这些输入属性进行预测。

假设程序的输入属性是JSON文件形式（也可当作队列中的JSON消息或JSON API响应），如下所示：

```
{
  "independent_variables": [
    {
      "name": "bmi",
      "value": 0.0616962065187
    },
    {
      "name": "ltg",
      "value": 0.0199084208763
    }
  ]
}
```

因此，在预测程序中创建如下结构类型，用于解码输入属性和编组输出预测：

```
// PredictionData includes the data necessary to make
// a prediction and encodes the output prediction.
type PredictionData struct {
    Prediction float64 `json:"predicted_diabetes_progression"`
    IndependentVars []IndependentVar `json:"independent_variables"`
}

// IndependentVar include information about and a
// value for an independent variable.
type IndependentVar struct {
    Name string `json:"name"`
    Value float64 `json:"value"`
}
```

创建一个函数，这个函数可以从模型输入目录（也就是持久化的模型）中读取`Model-Info`值进行预测。该`prediction`函数如下所示：

```
// Predict makes a prediction based on input JSON.
func Predict(modelInfo *ModelInfo, predictionData *PredictionData) error {

    // Initialize the prediction value
    // to the intercept.
    prediction := modelInfo.Intercept

    // Create a map of independent variable coefficients.
    coeffs := make(map[string]float64)
    varNames := make([]string, len(modelInfo.Coefficients))
    for idx, coeff := range modelInfo.Coefficients {
        coeffs[coeff.Name] = coeff.Coefficient
        varNames[idx] = coeff.Name
    }

    // Create a map of the independent variable values.
    varVals := make(map[string]float64)
    for _, indVar := range predictionData.IndependentVars {
        varVals[indVar.Name] = indVar.Value
    }

    // Loop over the independent variables.
    for _, varName := range varNames {

        // Get the coefficient.
        coeff, ok := coeffs[varName]
        if !ok {
            return fmt.Errorf("Could not find model coefficient %s",
varName)
        }

        // Get the variable value.
        val, ok := varVals[varName]
        if !ok {
            return fmt.Errorf("Expected a value for variable %s", varName)
        }

        // Add to the prediction.
        prediction = prediction + coeff*val
    }

    // Add the prediction to the prediction data.
    predictionData.Prediction = prediction
    return nil
}
```

这个预测函数和它的类型，允许方便地指定输入目录中 JSON 文件的任意属性，并且为每个文件输出疾病预测。该过程的代码实现如下：

```
// Declare the input and output directory flags.
inModelDirPtr := flag.String("inModelDir", "", "The directory containing
the model.")
inVarDirPtr := flag.String("inVarDir", "", "The directory containing the
input attributes.")
outDirPtr := flag.String("outDir", "", "The output directory")

// Parse the command line flags.
flag.Parse()
```

```
// Load the model file.
f, err := ioutil.ReadFile(filepath.Join(*inModelDirPtr, "model.json"))
if err != nil {
    log.Fatal(err)
}

// Unmarshal the model information.
var modelInfo ModelInfo
if err := json.Unmarshal(f, &modelInfo); err != nil {
    log.Fatal(err)
}

// Walk over files in the input.
if err := filepath.Walk(*inVarDirPtr, func(path string, info os.FileInfo,
err error) error {

    // Skip any directories.
    if info.IsDir() {
        return nil
    }

    // Open any files.
    f, err := ioutil.ReadFile(filepath.Join(*inVarDirPtr, info.Name()))
    if err != nil {
        return err
    }

    // Unmarshal the independent variables.
    var predictionData PredictionData
    if err := json.Unmarshal(f, &predictionData); err != nil {
        return err
    }

    // Make the prediction.
    if err := Predict(&modelInfo, &predictionData); err != nil {
        return err
    }

    // Marshal the prediction data.
    outputData, err := json.MarshalIndent(predictionData, "", "    ")
    if err != nil {
        log.Fatal(err)
    }

    // Save the marshalled output to a file.
    if err := ioutil.WriteFile(filepath.Join(*outDirPtr, info.Name()),
outputData, 0644); err != nil {
        log.Fatal(err)
    }

    return nil
}); err != nil {
    log.Fatal(err)
}
```

将使用与上一节中相同的 `Dockerfile` 和 `Makefile` 去 Docker 化这个预测程序。唯一不同的是这里将 Go 二进制文件重命名为 `goregpredict`，然后将 Docker

镜像标记为 dwhitena/goregpredict。使用 make 构建 Docker 镜像会返回类似前一节中的输出：

```
$ cat Makefile
all: compile docker push clean

compile:
        GOOS=linux GOARCH=amd64 CGO_ENABLED=0 go build -o goregpredict

docker:
        sudo docker build --force-rm=true -t dwhitena/goregpredict

push:
        sudo docker push dwhitena/goregpredict

clean:
        rm goregpredict
$ cat Dockerfile
FROM alpine
ADD goregpredict /
$ make
GOOS=linux GOARCH=amd64 CGO_ENABLED=0 go build -o goregpredict
sudo docker build --force-rm=true -t dwhitena/goregpredict
[sudo] password for dwhitena:
Sending build context to Docker daemon 2.38MB
Step 1/2 : FROM alpine
 ---> 7328f6f8b418
Step 2/2 : ADD goregpredict /
 ---> a2d9a63f4926
Removing intermediate container c1610b425835
Successfully built a2d9a63f4926
Successfully tagged dwhitena/goregpredict:latest
sudo docker push dwhitena/goregpredict
The push refers to a repository [docker.io/dwhitena/goregpredict]
77f12cb6c6d4: Pushed
5bef08742407: Layer already exists
latest: digest:
sha256:9a8a754c434bf2250b2f6102bb72d56fdf723f305aebcbf5bff7e5de707dd384
size: 738
3a05f65b1d1d: Layer already exists
5bef08742407: Layer already exists
ult: digest:
sha256:153adaa9b4b9a1f2bf02466201163c60230ae85164d9d22261f455979a94aed4
size: 738
rm goregpredict
```

9.1.2.3　本地测试 Docker 镜像

在推送 Docker 化的建模流程到任何服务器之前，最好在本地测试一下，确保看到的是预期行为。然后，一旦对那种行为满意了，便可以放心，这些 Docker 镜像在其他运行 Docker 的主机上行为是相同的。这一保证使得 Docker 镜像在维持可重复性的部署中发挥了重大作用。

假设在下列文件夹中有训练数据和一些输入属性文件：

```
$ ls
attributes model training
$ ls model
$ ls attributes
1.json 2.json 3.json
$ ls training
diabetes.csv
```

可以使用 docker run 命令将 Docker 镜像作为软件容器在本地运行。还可以使用 -v 标志挂载本地目录到正在运行的容器中。使得可以从 / 向本地文件系统读写文件。

首先，在 Docker 容器中运行单一回归模型训练，如下所示：

```
$ sudo docker run -v $PWD/training:/tmp/training -v $PWD/model:/tmp/model
dwhitena/goregtrain:single /goregtrain -inDir=/tmp/training -
outDir=/tmp/model

Regression Formula:
Predicted = 152.13 + bmi*949.44
```

现在，如果查看 model 目录，可以在 model.json 中看到新训练的模型系数和截距。它是从 Docker 镜像中运行的程序输出的。如下面代码所示：

```
$ ls model
model.json
$ cat model/model.json
{
    "intercept": 152.13348416289818,
    "coefficients": [
        {
            "name": "bmi",
            "coefficient": 949.4352603839862
        }
    ]
}
```

非常好，已经使用 Docker 容器训练了这个模型。现在利用这个模型来做预测。具体来说，使用训练好的模型来运行 goregpredict Docker 镜像，用于对 attributes 中的三个属性做出预测。在下面代码中，会看到属性文件在运行 Docker 镜像之前没有相应的预测，但是在运行 Docker 镜像之后就有了：

```
$ cat attributes/1.json
{
    "independent_variables": [
        {
            "name": "bmi",
            "value": 0.0616962065187
        },
        {
            "name": "ltg",
            "value": 0.0199084208763
```

仅供非商业用途或交流学习使用

```
        }
    ]
}
$ sudo docker run -v $PWD/attributes:/tmp/attributes -v
$PWD/model:/tmp/model dwhitena/goregpredict /goregpredict -
inModelDir=/tmp/model -inVarDir=/tmp/attributes -outDir=/tmp/attributes
$ cat attributes/1.json
{
    "predicted_diabetes_progression": 210.7100380636843,
    "independent_variables": [
        {
            "name": "bmi",
            "value": 0.0616962065187
        },
        {
            "name": "ltg",
            "value": 0.0199084208763
        }
    ]
}
```

9.1.2.4　在远端机器上运行 Docker 镜像

你可能会想，*自己构建和运行 Go 代码也能获得 Docker 一样的功能*。从本地来说是这样的，但是当想在除了笔记本以外的环境中运行相同功能时，Docker 的魔力就发生了。

在任何运行了 Docker 的主机上，可以用这些 Docker 镜像以完全相同的方式运行，并且它们会产生相同的行为。不必安装任何可能在 Docker 镜像层中已经存在的依赖。仅仅只需要 Docker。

猜猜结果如何？我们的 Docker 镜像仅仅只有 3 MB 左右！这意味着它们将以超快的速度下载到任何主机，启动和运行速度也超快。不必担心移动几个 GB 大小的虚拟机和手动指定资源。

9.2　构建可拓展和可重现的机器学习流水线

Docker 为部署机器学习流程到公司的基础架构中提供了很多途径，但是，仍有一些缺失的部分，如下所示：

- 如何将工作流程中的各个部分串联起来？在这个简单的示例中，有一个训练阶段和一个预测阶段。在其他流水线中，可能还会有数据预处理、数据拆分、数据组合、可视化、评估等。
- 如何让正确的数据进入工作流程的正确阶段，特别是接收到新数据和 / 或数据发生改变的时候。每当需要作出新的预测时，手动拷贝新属性到和预测图像相同位置的文件夹中是无法忍受的。不能每次更新训练数据集都要登录到

仅供非商业用途或交流学习使用

服务器上操作。

- 如何跟踪和重现工作流程的各种运行状况，用以维护、持续开发或者调试？如果随着时间推移做预测并且更新模型和 / 或训练集，需要了解哪些模型产生了哪些结果以保持可复现性（并且在某些情况下满足合规性）。
- 如何在多台机器甚至多个共享资源上扩展处理过程？可能需要在公司的一些共享资源上运行处理。当需要更多的计算能力和 / 或其他应用被调度到这些资源上的时候，如果能在这些资源上拓展处理过程就更好了。

庆幸的是，有一些开源工具（都是用 Go 编写的）可以为我们解决这些问题。不仅如此，这些工具还让我们能使用已经构建好的 Docker 镜像作为数据处理的主要单元。这些工具是 **Kubernetes**(k8s) 和 **Pachyderm**。

在第 1 章中已经接触到 Pachyderm。在那一章中，利用 Pachyderm 来说明数据版本的概念，并如你猜测的那样，Pachyderm 将有助于解决一些有关数据管理、追踪和复现的问题。Pachyderm 还将解决所有需要解决的可拓展性和流水线问题，因为 Pachyderm 提供了数据管理 / 版本控制的以及数据流水线的能力。

Kubernetes 是一个容器编排引擎，它在跨共享资源的集群上调度容器化的工作负载（例如我们的 Docker 镜像）中表现卓越。现在它非常受欢迎，Pachyderm 利用 Kubernetes 来管理容器化的数据处理流水线。

9.2.1 搭建 Pachyderm 和 Kubernetes 集群

因为 Kubernetes 可以部署在任何地方，因此运行在 Kubernetes 上的 Pachyderm 也能被部署到任何地方。你可以在任何流行的云服务提供商上部署 Pachyderm，甚至可以在自己的笔记本电脑上部署，部署 Pachyderm 集群的所需步骤如下所示：

1. 有一个运行着 Kubernetes 的集群。
2. 可以访问选择的对象存储（例如 S3、Glasglow Coma Scale（GCS）或 Minio），这个对象存储将为 Pachyderm 提供存储服务，用于存储所有版本化和处理后的数据。
3. 使用 Pachyderm CLI `pachctl` 在 Kubernetes 集群上部署 Pachyderm。

在任何云服务商或者企业内部服务器上部署 Pachyderm 的详细说明可以在此处找到：http://pachyderm.readthedocs.io/en/latest/deployment/deploy_intro.html。或者可以使用 `minikube` 轻松地实验和开发本地的 Pachyderm 集群，详情可参考：http://pachyderm.readthedocs.io/en/latest/getting_started/local_installation.html。

仅供非商业用途或交流学习使用

用如下命令可以看到 Kubernetes 集群运行状态（可以使用称为 kubectl 的 Kubernetes CLI 工具进行检查）：

```
$ kubectl get all
NAME READY STATUS RESTARTS AGE
po/etcd-2142892294-38ptw 1/1 Running 0 2m
po/pachd-776177201-04l6w 1/1 Running 0 2m

NAME CLUSTER-IP EXTERNAL-IP PORT(S) AGE
svc/etcd 10.0.0.141 <nodes> 2379:32379/TCP 2m
svc/kubernetes 10.0.0.1 <none> 443/TCP 3m
svc/pachd 10.0.0.178 <nodes> 650:30650/TCP,651:30651/TCP,652:30652/TCP 2m

NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE
deploy/etcd 1 1 1 1 2m
deploy/pachd 1 1 1 1 2m

NAME DESIRED CURRENT READY AGE
rs/etcd-2142892294 1 1 1 2m
rs/pachd-776177201 1 1 1 2m
```

在这里，可以看到 Pachyderm 守护进程（或服务）pachd 作为一个 pod 在 Kubernetes 集群中运行，它只是一个或多个容器的组合。流水线阶段也将作为 Kubernetes pod 运行，但不必担心太多，因为 Pachyderm 会处理细节。

以上输出中列出的端口和 IP 可能因部署和各种配置而不同。然而一个健康的 Pachyderm 集群应该看起来非常相似。

如果 Pachyderm 集群正在运行并且遵循了 Pachyderm 部署指南之一，那么还应该安装 pachctl CLI 工具。当 pachctl 成功连接到 Pachyderm 集群时，可以运行以下内容作为进一步的完整性检查（版本号可能会由于运行的 Pachyderm 版本不同而不一样）：

```
$ pachctl version
COMPONENT           VERSION
pachctl             1.6.0
pachd               1.6.0
```

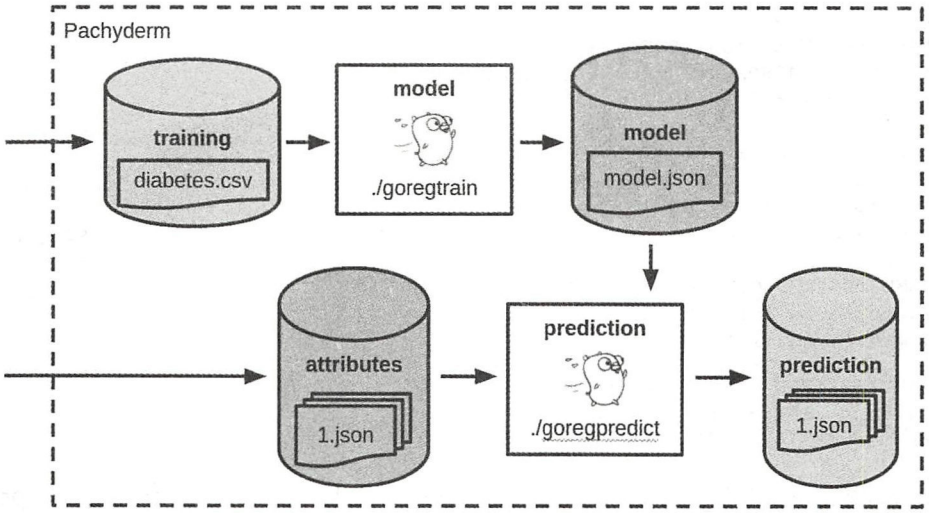
如果在部署 Pachyderm 时遇到困难，或者在构建或运行流水线时遇到任何困难，Pachyderm 社区拥有一个良好的公共 Pachyderm Slack 渠道。可以通过访问 http://slack.pachyderm.io/ 加入。这个社区每天都很活跃，欢迎在社区提出任何问题。

9.2.2　构建一个 Pachyderm 机器学习流水线

正如已经看到的，示例机器学习流水线有两个阶段。流水线的模型阶段将训练

仅供非商业用途或交流学习使用

模型并将此模型的持久版本导出到文件中。流水线的预测阶段将利用训练过的模型对输入属性文件进行预测。总的来说，这条流水线在 Pachyderm 中看起来如下：



上图中的每个柱面均代表版本化数据的 Pachyderm 数据仓库。已经在第 1 章中了解了这些数据仓库。每个框代表一个容器化的数据流水线阶段。Pachyderm 中的基本数据处理单元是 Docker 镜像。因此，可以利用在数据流水线的前面部分所创建的 Docker 镜像。

通过将版本化的数据集合（再次将它们视为数据的一种 Git），使用容器处理这些数据集合以及将结果保存到其他版本化的数据集合中，Pachyderm 流水线具有一些非常有趣和有用的含义。

首先，可以随时回撤到任意版本，以便及时查看数据在那个时间点的任何部分的状态。如果想要开发数据的某种状态，这将有助于进一步开发模型。它也可以通过跟踪团队的时序数据帮助推进开发团队的协作。并非所有对数据的更改都是好的，这需要系统具有恢复不良和损坏数据提交的能力。

接下来，所有的流水线结果都与产生这些结果的 Docker 镜像和其他数据的状态相关联。Pachyderm 团队称这为**起源**（provenance）。这是理解哪些处理过程和哪些数据对特定结果有贡献的能力。例如，通过这个流水线，可以确定产生特定结果的持久模型的确切版本，以及生成该模型的确切 Docker 镜像，还有输入到那个确切 Docker 镜像的数据的确切状态。因此，能够准确再现任何流水线的整个运行过程，调试异常的模型行为并将结果关联于相应的输入数据。

最后，由于 Pachyderm 知道数据仓库的哪些部分是新的或不同的，数据流水线可以增量地处理数据并可以自动触发。假设已经向 `attributes` 数据仓库提

仅供非商业用途或交流学习使用

交了一百万个属性文件，然后提交了另外十个属性文件。无须处理这前一百万条数据用以更新结果，Pachyderm 知道只需要处理这十个新文件，并将它们发送到 prediction 阶段进行处理。此外，Pachyderm 将（默认情况下）自动触发此更新，因为它知道需要更新以保持结果与输入同步。

增量处理和自动触发是 Pachyderm 流水线的一些默认行为，但它们不是 Pachyderm 唯一可做的事情。事实上，这里仅仅是肤浅地谈了谈 Pachyderm 的可能性。还可以执行分布式的 map/reduce 式操作、分布式图像处理、数据库表的定期处理等。因此，这里的技术可以转移到各种领域。

9.2.2.1　创建并填充输入数据仓库

要在部署的 Pachyderm 集群上创建 Pachyderm 流水线，首先需要为流水线创建输入储存库。请记住，流水线具有 training 和 attributes 数据仓库，可以驱动流水线的其余部分。当将数据放入这些数据流水线时，Pachyderm 将触发流水线的下游部分并计算结果。

在第 1 章中看到，如何在 Pachyderm 中创建数据仓库并使用 pachctl 将数据提交到这些数据仓库，这里将尝试直接从 Go 程序中做此操作。在这个特定的例子中，这样做没有任何实际的好处，因为训练和测试数据已经在文件中了，可以很容易地通过名称将这些文件提交到 Pachyderm。

但是，想象一个更现实的场景。可能希望将现有数据流水线集成到公司的一些其他 Go 服务上。其中一项服务可能会处理来自医生或诊所传入的患者医疗属性，并且希望将这些属性提供给数据流水线，以便数据流水线可以对疾病进展做出相应的预测。

在这种情况下，希望将属性从服务中直接传递给 Pachyderm，而不是手动复制所有数据。这是 Pachyderm Go 客户端 github.com/pachyderm/pachyderm/src/client 可以派上用场的地方。使用 Pachyderm Go 客户端，可以按如下方式创建输入数据仓库（在连接到 Pachyderm 集群后）：

```
// Connect to Pachyderm using the IP of our
// Kubernetes cluster. Here we will use localhost
// to mimic the scenario when you have k8s running
// locally and/or you are forwarding the Pachyderm
// port to your localhost.. By default
// Pachyderm will be exposed on port 30650.
c, err := client.NewFromAddress("0.0.0.0:30650")
if err != nil {
    log.Fatal(err)
}
defer c.Close()
```

仅供非商业用途或交流学习使用

```
// Create a data repository called "training."
if err := c.CreateRepo("training"); err != nil {
    log.Fatal(err)
}

// Create a data repository called "attributes."
if err := c.CreateRepo("attributes"); err != nil {
    log.Fatal(err)
}

// Now, we will list all the current data repositories
// on the Pachyderm cluster as a sanity check. We
// should now have two data repositories.
repos, err := c.ListRepo(nil)
if err != nil {
    log.Fatal(err)
}

// Check that the number of repos is what we expect.
if len(repos) != 2 {
    log.Fatal("Unexpected number of data repositories")
}

// Check that the name of the repo is what we expect.
if repos[0].Repo.Name != "attributes" || repos[1].Repo.Name != "training" {
    log.Fatal("Unexpected data repository name")
}
```

编译并运行它会按预期创建 repos，可用如下内容确认：

```
$ go build
$ ./myprogram
$ pachctl list-repo
NAME CREATED SIZE
attributes 3 seconds ago 0B
training 3 seconds ago 0B
```

为了简单起见，上面的代码只是创建并检查 repo 一次。如果再次运行代码，则会出现错误，因为已经创建了 repo。为了改善这一点，可以检查 repo 是否存在，如果不存在则创建它。也可以通过 repo 名称来检查 repo 是否存在。

现在继续往下讲解，将属性文件放入 attributes 数据仓库，将 diabetes.csv 训练数据集放入 training 数据仓库。这也可以通过 Pachyderm 客户端直接在 Go 中完成：

```
// Connect to Pachyderm.
c, err := client.NewFromAddress("0.0.0.0:30650")
if err != nil {
    log.Fatal(err)
}
defer c.Close()

// Start a commit in our "attributes" data repo on the "master" branch.
commit, err := c.StartCommit("attributes", "master")
```

仅供非商业用途或交流学习使用

```
if err != nil {
    log.Fatal(err)
}

// Open one of the attributes JSON files.
f, err := os.Open("1.json")
if err != nil {
    log.Fatal(err)
}

// Put a file containing the attributes into the data repository.
if _, err := c.PutFile("attributes", commit.ID, "1.json", f); err != nil {
    log.Fatal(err)
}

// Finish the commit.
if err := c.FinishCommit("attributes", commit.ID); err != nil {
    log.Fatal(err)
}

// Start a commit in our "training" data repo on the "master" branch.
commit, err = c.StartCommit("training", "master")
if err != nil {
    log.Fatal(err)
}

// Open up the training data set.
f, err = os.Open("diabetes.csv")
if err != nil {
    log.Fatal(err)
}

// Put a file containing the training data set into the data repository.
if _, err := c.PutFile("training", commit.ID, "diabetes.csv", f); err !=
nil {
    log.Fatal(err)
}

// Finish the commit.
if err := c.FinishCommit("training", commit.ID); err != nil {
    log.Fatal(err)
}
```

运行它确保将数据提交到适当数据仓库中的 Pachyderm 里，如下所示：

```
$ go build
$ ./myprogram
$ pachctl list-repo
NAME CREATED SIZE
training 13 minutes ago 73.74KiB
attributes 13 minutes ago 210B
$ pachctl list-file training master
NAME TYPE SIZE
diabetes.csv file 73.74KiB
$ pachctl list-file attributes master
NAME TYPE SIZE
1.json file 210B
```

仅供非商业用途或交流学习使用

完成上面提到的提交很重要。如果留下一个孤立的提交任由其挂起，可能会阻止其他数据的提交。因此，可以考虑推迟提交的完成，以保证安全。

好！现在远程 Pachyderm 集群上有正确的输入数据仓库中的数据。接下来，可以实际处理这些数据并生成结果。

9.2.2.2 创建并运行处理阶段

处理阶段通过 Pachyderm 中的**流水线规范**（pipeline specification）以声明方式创建。如果之前曾使用过 Kubernetes，则应该对此类交互很熟悉。基本上，向 Pachyderm 声明想要进行的处理，然后 Pachyderm 会处理所有的细节，确保像定义那样执行。

首先，使用存储在 `train.json` 中的 JSON 流水线规范创建流水线的 `model` 阶段。这个 JSON 规范如下所示：

```
{
  "pipeline": {
    "name": "model"
  },
  "transform": {
    "image": "dwhitena/goregtrain:single",
    "cmd": [
      "/goregtrain",
      "-inDir=/pfs/training",
      "-outDir=/pfs/out"
    ]
  },
  "parallelism_spec": {
    "constant": "1"
  },
  "input": {
    "atom": {
      "repo": "training",
      "glob": "/"
    }
  }
}
```

这看起来有点复杂，但实际上这里只发生了不多的事情。接下来会剖析规范以加深了解：

- 首先，告诉 Pachyderm 来命名流水线 `model`。
- 接下来，告诉 Pachyderm 希望这个 `model` 流水线使用 `dwhitena/goregtrain:single` 来处理数据，并且希望流水线在处理数据时在容器中运行 `goregtrain` 二进制文件。
- 最后，规范告诉 Pachyderm 打算将 `training` 数据仓库作为输入进行处理。

为了理解规范的其他一些细节，需要首先讨论当在 Pachyderm 集群上使用规范创建流水线时会发生什么。对于这种情况，Pachyderm 将使用 Kubernetes

在 Kubernetes 群集上安排一个或多个工作 pod。这些工作 pod 随即会等待，准备处理由 Pachyderm 守护程序提供给他们的任何数据。当需要在输入数据仓库（training）中处理数据时，Pachyderm 守护进程将触发这些工作 pod 处理数据的工作。Pachyderm 会自动将输入数据挂载在 /pfs/<input repo name>（在本例中为 /pfs/training）的容器中。容器写入 /pfs/out 的数据将自动在与流水线（本例中为 model）同名的输出数据仓库中进行版本化。

规范中的 parallelism_spec 部分告诉 Pachyderm 应该启动多少工作节点来处理输入数据。在这里，将启动一个工作节点并连续处理数据。在本章的后面，将讨论并行化的流水线和相关的数据分片，这些分片由规范 input 部分的 glob 模式控制。

这么描述已经足够了，下面在 Pachyderm 集群上创建这个 model 流水线阶段，并开始处理一些数据。流水线可以按如下方法简单地创建：

```
$ pachctl create-pipeline -f model.json
```

当这样做时，可以看到 Pachyderm 使用 kubectl 在 Kubernetes 集群上创建的工作节点，如下面的代码所示：

```
$ kubectl get pods
NAME READY STATUS RESTARTS AGE
etcd-2142892294-38ptw 1/1 Running 0 2h
pachd-776177201-04l6w 1/1 Running 0 2h
pipeline-model-v1-p0lnf 2/2 Running 0 1m
```

也会看到 Pachyderm 已经自动触发了一个任务来执行模型训练。请记住，发生这种情况是因为 Pachyderm 知道输入数据仓库（training）中存在尚未处理的数据。可以看到触发的任务及其结果如下：

```
$ pachctl list-job
ID OUTPUT COMMIT STARTED DURATION RESTART PROGRESS DL UL STATE
14f052ae-878d-44c9-a1f9-ab0cf6d45227 model/a2c7b7dfb44a40e79318c2de30c7a0c8
3 minutes ago Less than a second 0 1 + 0 / 1 73.74KiB 160B success
$ pachctl list-repo
NAME CREATED SIZE
model 3 minutes ago 160B
training About an hour ago 73.74KiB
attributes About an hour ago 210B
$ pachctl list-file model master
NAME TYPE SIZE k8s
model.json file 160B
$ pachctl get-file model master model.json
{
    "intercept": 152.13348416289818,
    "coefficients": [
        {
            "name": "bmi",
            "coefficient": 949.4352603839862
        }
    ]
}
```

太好了！可以看到，模型训练产生了预期的结果。然而，Pachyderm足够的智能，可以通过触发来启动流水线，每当修改训练数据集时它都会更新这个模型。

prediction流水线可以以类似的方式创建。其流水线规范pipeline.json如下所示：

```
{
  "pipeline": {
    "name": "prediction"
  },
  "transform": {
    "image": "dwhitena/goregpredict",
    "cmd": [
      "/goregpredict",
      "-inModelDir=/pfs/model",
      "-inVarDir=/pfs/attributes",
      "-outDir=/pfs/out"
    ]
  },
  "parallelism_spec": {
    "constant": "1"
  },
  "input": {
    "cross": [
      {
        "atom": {
          "repo": "attributes",
          "glob": "/*"
        }
      },
      {
        "atom": {
          "repo": "model",
          "glob": "/"
        }
      }
    ]
  }
}
```

这个流水线规范的主要区别是有两个输入数据仓库，而不是一个。这两个输入数据仓库是以前创建的attributes数据仓库和包含model流水线输出的model数据仓库。这些使用cross图元进行组合。cross确保处理所有相关的模型和属性文件对。

如果有兴趣，可以找到更多在Pachyderm文档中关于合并Pachyderm数据的其他方法，请访问http://pachyderm.readthedocs.io/en/latest/。

创建该流水线并检查结果（对应于1.json），可以按如下方式完成：

```
$ pachctl create-pipeline -f prediction.json
$ pachctl list-job
ID OUTPUT COMMIT STARTED DURATION RESTART PROGRESS DL UL STATE
03f36398-89db-4de4-ad3d-7346d56883c0
prediction/5ce47c9e788d4893ae00c7ee6b1e8431 About a minute ago Less than a
second 0 1 + 0 / 1 370B 266B success
14f052ae-878d-44c9-a1f9-ab0cf6d45227 model/a2c7b7dfb44a40e79318c2de30c7a0c8
19 minutes ago Less than a second 0 1 + 0 / 1 73.74KiB 160B success
$ pachctl list-repo
NAME CREATED SIZE
prediction About a minute ago 266B
model 19 minutes ago 160B
training About an hour ago 73.74KiB
attributes About an hour ago 210B
$ pachctl list-file prediction master
NAME TYPE SIZE
1.json file 266B
$ pachctl get-file prediction master 1.json
{
    "predicted_diabetes_progression": 210.7100380636843,
    "independent_variables": [
        {
            "name": "bmi",
            "value": 0.0616962065187
        },
        {
            "name": "ltg",
            "value": 0.0199084208763
        }
    ]
}
```

可以看到创建了另一个数据仓库 prediction，以对 prediction 流水线的输出进行版本化。通过这种方式，Pachyderm 逐渐建立输入数据仓库、数据处理阶段和输出数据仓库之间的链接，以便它始终知道链接的数据和处理组件。

完整的机器学习流水线现在已经部署，流水线的每个阶段已经运行了一次。然而，流水线已准备就绪，正在等待处理更多数据。这里所要做的就是将数据放在流水线的顶部，Pachyderm 将自动触发任何需要进行的下游处理来更新结果。接下来通过将两个以上的文件提交到 attributes 数据仓库来说明这一点，如下所示：

```
$ ls
2.json 3.json
$ pachctl put-file attributes master -c -r -f .
$ pachctl list-file attributes master
NAME TYPE SIZE
1.json file 210B
2.json file 211B
3.json file 211B
```

这将自动触发另一个预测任务来更新结果。新任务可以在以下代码中看到：

```
$ pachctl list-job
ID OUTPUT COMMIT STARTED DURATION RESTART PROGRESS DL UL STATE
be71b034-9333-4f75-b443-98d7bc5b48ab
prediction/bb2fd223df664e70ad14b14491109c0f About a minute ago Less than a
second 0 2 + 1 / 3 742B 536B success
03f36398-89db-4de4-ad3d-7346d56883c0
prediction/5ce47c9e788d4893ae00c7ee6b1e8431 8 minutes ago Less than a
second 0 1 + 0 / 1 370B 266B success
14f052ae-878d-44c9-a1f9-ab0cf6d45227 model/a2c7b7dfb44a40e79318c2de30c7a0c8
26 minutes ago Less than a second 0 1 + 0 / 1 73.74KiB 160B success
```

另外，可以看到 prediction 库有两个新的结果，它们是在两个新的输入文件上再次运行预测代码得到的：

```
$ pachctl list-file prediction master
NAME TYPE SIZE
1.json file 266B
2.json file 268B
3.json file 268B
```

注意 Pachyderm 在这里没有重新处理 1.json，尽管它显示了最新的结果。在引擎内部逻辑中，Pachyderm 知道在 attributes 中已经有对应于 1.json 的结果，所以不需要重新处理它。

9.2.3 更新流水线并检查出处

随着时间的推移，可能会更新机器学习模型。他们正在处理的数据可能随着时间而改变，或者刚刚开发了一个不同的或更好的模型。无论如何，很可能需要更新流水线。

假设流水线处于上一节描述的状态，已经创建了完整的流水线，训练流程已经执行完毕，并且已经将两个提交处理到 attributes 数据仓库中。现在，让我们更新模型流水线以训练多元回归。

现在，更新模型流水线来训练的是多元回归模型，而不是单一回归模型。这其实很简单，只需要将 model.json 流水线规范中的 "image":"dwhitena/goregtrain:single" 更改为 "image":"dwhitena/goregtrain:multi"，然后更新流水线，如下所示：

```
$ pachctl update-pipeline -f model.json --reprocess
```

当给 Pachyderm 这个新的规范时，Pachyderm 会自动更新工作 pod，以便它们使用多元回归模型运行容器。此外，因为我们提供了 --reprocess 标志，Pachyderm 将触发一个或多个新作业，以重新处理任何使用旧镜像处理过的新镜像的输入提交。如下是这些重新处理的任务：

```
$ pachctl list-job
ID OUTPUT COMMIT STARTED DURATION RESTART PROGRESS DL UL STATE
03667301-980b-484b-afbe-7ea30da695f5
prediction/ef32a74b04ee4edda7bc2a2b469f3e03 2 minutes ago Less than a
second 0 3 + 0 / 3 1.355KiB 803B success
5587e13c-4854-4f3a-bc4c-ae88c65c007f model/f54ca1a0142542579c1543e41f5e9403
2 minutes ago Less than a second 0 1 + 0 / 1 73.74KiB 252B success
be71b034-9333-4f75-b443-98d7bc5b48ab
prediction/bb2fd223df664e70ad14b14491109c0f 16 minutes ago Less than a
second 0 2 + 1 / 3 742B 536B success
03f36398-89db-4de4-ad3d-7346d56883c0
prediction/5ce47c9e788d4893ae00c7ee6b1e8431 23 minutes ago Less than a
second 0 1 + 0 / 1 370B 266B success
14f052ae-878d-44c9-a1f9-ab0cf6d45227 model/a2c7b7dfb44a40e79318c2de30c7a0c8
41 minutes ago Less than a second 0 1 + 0 / 1 73.74KiB 160B success
```

看看这个输出结果，可以注意到一些有趣且非常有用的东西。当在 model 阶段训练新多元回归模型并将其输出为 model.json 时，Pachyderm 发现 prediction 流水线的结果现在与最新模型不同步。因此，Pachyderm 自动触发另一项工作，使用新的多元回归模型更新以前的结果。

这个工作流程对于管理已部署的模型以及正在开发的模型都非常有用，因为无须手动对一堆模型版本和相关数据进行重置。一切都自动发生。然而，这个更新产生了一个自然的问题。如何知道哪些模型产生了哪些结果？

这就是 Pachyderm 的数据版本与流水线相结合的地方。可以查看任何预测结果并检查该提交以确定提交的来源。如下所示：

```
$ pachctl inspect-commit prediction ef32a74b04ee4edda7bc2a2b469f3e03
Commit: prediction/ef32a74b04ee4edda7bc2a2b469f3e03
Parent: bb2fd223df664e70ad14b14491109c0f
Started: 8 minutes ago
Finished: 8 minutes ago
Size: 803B
Provenance: training/e0f9357455234d8bb68540af1e8dde81
attributes/2e5fef211f14498ab34c0520727296bb
model/f54ca1a0142542579c1543e41f5e9403
```

在这种情况下，我们可以看到由 f54ca1a0142542579c1543e41f5e9403 模型得出的预测结果 ef32a74b04ee4edda7bc2a2b469f3e03，可以在这里找到：

```
$ pachctl get-file model f54ca1a0142542579c1543e41f5e9403 model.json
{
    "intercept": 152.13348416289583,
    "coefficients": [
        {
            "name": "bmi",
            "coefficient": 675.069774431606
        },
        {
            "name": "ltg",
            "coefficient": 614.9505047824742
        }
    ]
}
```

但是，如果看一下之前的预测，会看到它是由不同的模型产生的：

```
$ pachctl inspect-commit prediction bb2fd223df664e70ad14b14491109c0f
Commit: prediction/bb2fd223df664e70ad14b14491109c0f
Parent: 5ce47c9e788d4893ae00c7ee6b1e8431
Started: 25 minutes ago
Finished: 25 minutes ago
Size: 802B
Provenance: model/a2c7b7dfb44a40e79318c2de30c7a0c8
training/e0f9357455234d8bb68540af1e8dde81
attributes/2e5fef211f14498ab34c0520727296bb
$ pachctl get-file model a2c7b7dfb44a40e79318c2de30c7a0c8 model.json
{
    "intercept": 152.13348416289818,
    "coefficients": [
        {
            "name": "bmi",
            "coefficient": 949.4352603839862
        }
    ]
}
```

无论有多少工作正在运行，无论流水线更新了多少次，都可以追溯流水线中任何数据的来源。这在创建可持续的机器学习工作流程中非常重要，这些工作流程可以随时间维护，改进和更新，而不会带来重大的麻烦。

9.2.4 缩放流水线阶段

Pachyderm 中的每个流水线规范都可以有相应的 parallelism_spec 字段。该字段与输入中的 glob 模式一起，可在输入数据上并行流水线阶段。每个流水线阶段都可以独立地进行缩放，不用依赖任何其他流水线阶段。

流水线规范中的 parallelism_spec 使得可以控制流水线阶段启动多少 Pachyderm 工作节点去处理数据。例如，下面的 parallelism_spec 会告诉 Pachyderm 为一个流水线启动 10 个工作节点：

```
"parallelism_spec": {
  "constant": "10"
},
```

输入中的 glob 模式告诉 Pachyderm，如何通过 parallelism_spec 声明的工作节点去共享输入数据。例如，glob 模式中 /* 会告诉 Pachyderm，它可以将输入数据拆分成多份文件，这些文件会直接保存到数据仓库的根目录里。/*/* 的 glob 模式会告诉 Pachyderm，它可以将数据拆分为包含文件或目录的数据，这些文件或目录位于数据仓库目录结构的更深层。如果对 glob 模式不熟悉，请参阅以下示例中的说明：https://en.wikipedia.org/wiki/Glob_(programming)。

就示例流水线而言，可以想象为我们需要扩展流水线的 prediction 阶段，因

为可以看到大量需要预测的属性涌入。对于这种情况，可以将 prediction.json 流水线规范中的 parallelism_spec 更改为 "constant":"10"。一旦用新规格更新流水线，Pachyderm 将为 prediction 流水线启动 10 个新的工作节点，如下所示：

```
$ pachctl update-pipeline -f prediction.json
 $ kubectl get pods
 NAME READY STATUS RESTARTS AGE
 etcd-2142892294-38ptw 1/1 Running 0 3h
 pachd-776177201-04l6w 1/1 Running 0 3h
 pipeline-model-v2-168k5 2/2 Running 0 29m
 pipeline-prediction-v2-0p6zs 0/2 Init:0/1 0 6s
 pipeline-prediction-v2-3fbsc 0/2 Init:0/1 0 6s
 pipeline-prediction-v2-c3m4f 0/2 Init:0/1 0 6s
 pipeline-prediction-v2-d11b9 0/2 Init:0/1 0 6s
 pipeline-prediction-v2-hjdll 0/2 Init:0/1 0 6s
 pipeline-prediction-v2-hnk64 0/2 Init:0/1 0 6s
 pipeline-prediction-v2-q29f1 0/2 Init:0/1 0 6s
 pipeline-prediction-v2-qlhm4 0/2 Init:0/1 0 6s
 pipeline-prediction-v2-qrnf9 0/2 Init:0/1 0 6s
 pipeline-prediction-v2-rdt81 0/2 Init:0/1 0 6s
 $ kubectl get pods
 NAME READY STATUS RESTARTS AGE
 etcd-2142892294-38ptw 1/1 Running 0 3h
 pachd-776177201-04l6w 1/1 Running 0 3h
 pipeline-model-v2-168k5 2/2 Running 0 30m
 pipeline-prediction-v2-0p6zs 2/2 Running 0 26s
 pipeline-prediction-v2-3fbsc 2/2 Running 0 26s
 pipeline-prediction-v2-c3m4f 2/2 Running 0 26s
 pipeline-prediction-v2-d11b9 2/2 Running 0 26s
 pipeline-prediction-v2-hjdll 2/2 Running 0 26s
 pipeline-prediction-v2-hnk64 2/2 Running 0 26s
 pipeline-prediction-v2-q29f1 2/2 Running 0 26s
 pipeline-prediction-v2-qlhm4 2/2 Running 0 26s
 pipeline-prediction-v2-qrnf9 2/2 Running 0 26s
 pipeline-prediction-v2-rdt81 2/2 Running 0 26s
```

现在，任何提交到 attributes 数据仓库中的属性都会有 10 个工作节点并行处理。例如，如果我们将 100 个以上的属性文件添加到 attributes 中，则 Pachyderm 会将这些文件的 1/10 发送给 10 个工作节点中的每一个去并行处理。所有的结果仍然可以在 prediction 数据仓库中以相同方式看到。

当然，设置一定数量的工作节点并不是唯一可以用 Pachyderm 和 Kubernetes 进行缩放的方法。Pachyderm 也可以让你通过为闲置的工作节点设置一个缩减阈值来减少工作节点规模。此外，Pachyderm 工作节点的自动缩放可以同 Kubernetes 集群资源的自动缩放相结合，用以应对突发数据和批处理需求。此外，Pachyderm 还允许你指定流水线所需的资源，使用此资源规范可以将机器学习工作负载转移到加速器（例如 GPU）。

仅供非商业用途或交流学习使用

9.3 参考书目

Docker：

- Docker 简介：https://training.docker.com/introduction-to-docker
- 为 Go 应用构建最小化的 Docker 镜像：https://blog.codeship.com/building-minimal-docker-containers-for-go-applications/

Pachyderm：

- 公共 Slack 频道：http://slack.pachyderm.io/
- 入门指南：http://pachyderm.readthedocs.io/en/latest/getting_started/getting_started.html
- Go 客户端文档：https://godoc.org/github.com/pachyderm/pachyderm/src/client
- 用 Pachyderm 进行机器学习：http://pachyderm.readthedocs.io/en/latest/cookbook/ml.html
- 机器学习示例：http://pachyderm.readthedocs.io/en/latest/examples/readme.html#machine-learning
- 使用 Pachyderm 进行分布式处理：http://pachyderm.readthedocs.io/en/latest/fundamentals/distributed_computing.html
- 常见 Pachyderm 部署：http://pachyderm.readthedocs.io/en/latest/deployment/deploy_intro.html
- 自动缩放 Pachyderm 群集：http://pachyderm.readthedocs.io/en/latest/managing_pachyderm/autoscaling.html

9.4 小结

从仅收集 CSV 数据到完全部署的分布式机器学习流水线，我们做到了！现在可以使用 Go 构建机器学习模型，并在一组机器上放心地部署这些模型。这些模式会允许你构建和部署各种各样的智能应用程序，我迫不及待地想知道你创建的内容！请保持联系。

仅供非商业用途或交流学习使用

附录

与机器学习相关的算法 / 技术

有一些算法是跟本书前面无法详述的示例相关，在这里会逐一解决。

梯度下降

在许多示例中（包括第 4 章和第 5 章中的示例）都采用了称为**梯度下降**的优化技术。梯度下降法有多种变体，一般来说，在机器学习的世界中它们无所不在。最显著的用法是，它们被用于确定线性或逻辑回归等算法的最优系数。因此也会在更复杂的技术中起作用，这些技术至少是部分基于线性 / 逻辑回归的算法（例如神经网络）。

梯度下降法的总体思路是确定某些参数的变化方向和幅度，这使得你可以向正确的方向移动以优化某些度量（例如误差）。可以想象一下站在某个观光点，如果要向低海拔地区移动，那么就需要采取方向是向下的步伐。这基本上也是梯度下降算法在优化参数时所做的。

下面来看看所谓**随机梯度下降**（SGD），用以获得关于这个过程的更多直观信息。这是一种递增梯度下降。实际上在第 5 章中实施逻辑回归时就用到了 SGD。在那个例子中，实施了逻辑回归参数的训练或拟合，如下所示：

```
// logisticRegression fits a logistic regression model
// for the given data.
func logisticRegression(features *mat64.Dense, labels []float64, numSteps
int, learningRate float64) []float64 {

    // Initialize random weights.
    _, numWeights := features.Dims()
    weights := make([]float64, numWeights)

    s := rand.NewSource(time.Now().UnixNano())
    r := rand.New(s)
```

仅供非商业用途或交流学习使用

```
    for idx, _ := range weights {
        weights[idx] = r.Float64()
    }

    // Iteratively optimize the weights.
    for i := 0; i < numSteps; i++ {

        // Initialize a variable to accumulate error for this iteration.
        var sumError float64

        // Make predictions for each label and accumulate error.
        for idx, label := range labels {

            // Get the features corresponding to this label.
            featureRow := mat64.Row(nil, idx, features)

            // Calculate the error for this iteration's weights.
            pred := logistic(featureRow[0]*weights[0] +
featureRow[1]*weights[1])
            predError := label - pred
            sumError += math.Pow(predError, 2)

            // Update the feature weights.
            for j := 0; j < len(featureRow); j++ {
                weights[j] += learningRate * predError * pred * (1 - pred
* featureRow[j]
            }
        }
    }
    return weights
}
```

在注释“//Iteratively optimize the weights”下面的循环中，实现了SGD以优化逻辑回归参数。这里单独拿出这个循环来分析一下究竟发生了什么。

首先，利用当前权重及预测值和理想值（即实际观察值）之间的差异来计算模型的输出：

```
// Calculate the error for this iteration's weights.
pred := logistic(featureRow[0]*weights[0] + featureRow[1]*weights[1])
predError := label - pred
sumError += math.Pow(predError, 2)
```

然后，根据SGD，按照以下定义计算参数（在这种情况下是weights）的更新：

$$update=learning\ rate \times gradient\ of\ the\ parameters$$

gradient是本问题中成本函数的数学梯度。

关于这个量的更详细数学信息可以在这里找到：http://mathworld.wolfram.com/Gradient.html

然后可以将更新应用到参数中，如下所示：

$$perameters=parameters-update$$

仅供非商业用途或交流学习使用

在这个逻辑回归中，可以得到以下形式：

```
// Update the feature weights.
for j := 0; j < len(featureRow); j++ {
    weights[j] += learningRate * predError * pred * (1 - pred) *
featureRow[j]
}
```

这种类型的 SGD 广泛应用于机器学习中。然而，某些情况下这种梯度下降会导致过度拟合或者陷入局部最小值 / 最大值（而不是寻找全局最优值）

为了解决这些问题，可以使用梯度下降的变体，也就是所谓的**分批梯度下降**。在分批梯度下降中，可以根据所有训练集中的梯度计算每个参数更新，而不是根据特定观测或者数据集中某一行的梯度。这种方法可以避免过度拟合，但是它可能相当缓慢并且存在内存问题。因为需要针对每个参数的整个数据集计算梯度。**小批量梯度下降**是另一种变体，它试图保持分批梯度下降的一些优点，同时在计算上更容易处理。在小批量梯度下降中，梯度是基于训练数据集的子集，而不是整个训练集计算。

在逻辑回归情况下，能看到使用梯度上升或下降。其中梯度上升除了应用于成本函数的负数，剩下的与梯度下降是相同的。在这里会有进一步讨论：https://stats.stackexchange.com/questions/261573/using-gradient-ascent-instead-of-gradient-descent-for-logistic-regression。

gonum.org/v1/gonum/optimize 已经实现了梯度下降的方法。更多详细信息请参阅文档：https://godoc.org/gonum.org/v1/gonum/optimize#GradientDescent。

熵、信息增益及相关方法

在第 5 章中讨论了决策树方法，其中模型是由 if/then 语句树组成。这些决策树的 if/then 部分基于训练集的特征之一来分割预测逻辑。在尝试将医疗患者分类为不健康或者健康类别的一个示例中，决策树可能首先基于性别特征进行分割，然后基于年龄特征，再然后基于权重特征等，最后得到健康或不健康。

在决策树中算法如何选择首先使用哪些特征呢？在前面的示例中，是先根据性别或者权重分割，然后再基于其他特征分割。这里需要有一种方法能以最佳方式安排分割，以便模型能做出最好的预测。

包括第 5 章中用到的，许多决策树模型实现都使用熵和信息增益的分析来构建决策树。来看一个例子用以说明该过程，假设有以下人们健康状况的数量以及这些

仅供非商业用途或交流学习使用

人的各种特征数据。

	Healthy	Unhealthy
Vegan Diet	5	2
Vegetarian Diet	4	1
Meat Eating Diet	3	4

	Healthy	Unhealthy
Age 40+	3	5
Age <40	9	2

这里的数据有两个特征：饮食（Diet）和年龄（Age），希望建立一个基于饮食和年龄的决策树来预测人们是否健康。为了做到这一点，首先需要决定是先基于年龄还是先基于饮食分割决策树。请注意，数据中显示有 12 位健康人士和 7 位不健康人士。

首先，将计算数据中类别的总体状况或者总熵，定义如下：

$$E=-p_1\log_2(p_1)-p_2\log_2(p_2)-p_3\log_2(p_3)-\cdots$$

在这里，p_1，p_2 等是第一类别、第二类别的概率。在本例中（因为有 12 位健康人士和 7 位不健康人士），总熵如下：

$$E(Health)=-\left(\frac{12}{19}\right)\log_2\left(\frac{12}{19}\right)-\left(\frac{7}{19}\right)\log_2\left(\frac{7}{19}\right)=0.95$$

这个 *0.95* 测量值代表了健康数据的同质性。它介于 0 和 1 之间，数值越高，说明数据越不均匀。

为了确定是应该先在基于年龄还是先基于饮食分割树，将计算哪些特征提供了最多的信息增益。简而言之，发现这样的特征，可以在分割特征后获得最大的同质性，如前面的熵所测量的那样。熵的减少被称为**信息增益**。

在示例中某个特征的信息增益定义如下：

$$G=E(Health)-E(Health, Feature)$$

这里，$E(Health, Feature)$ 是关于给定特征（*Age* 或者 *Diet*）的熵的第二测量。对于饮食来说，第二项测量计算如下：

$$E(Health, Age)=p_{40+}E(Health, 40+)+p_{<40}E(Health, <40)$$

$p40+$ 和 $p<40$ 的数量是年龄为 *40+* 或 *<40* 的概率（分别为 8/19 和 11/19）。$E(Health, 40+)$ 和 $E(Health, <40)$ 的数量是健康熵（如前述公式中所定义的），但是仅使用了 40 岁以上和 40 岁以下的计数。

对于示例数据，年龄特征的信息增益为 *0.152*，饮食信息增益为 *0.079*。因此，

仅供非商业用途或交流学习使用

会首先选择基于年龄特征分割决策树，因为它最大限度地增加了数据的整体同质性。

可以在 http://www.saedsayad.com/decision_tree.htm 找到更多关于基于熵构建决策树的信息，可以在 https://github.com/sjwhitworth/golearn/blob/master/trees/entropy.go 中查看 Go 中的示例实现。

反向传播

第 8 章包括一个从头开始构建的神经网络的例子。这个神经网络包括一个反向传播方法的实现用来训练神经网络，它可以在任何神经网络代码中找到。在本章中讨论了一些细节。但是，这种方法应用很广泛，所以想在这里一步一步地理解它。

为了训练一个带有反向传播的神经网络，针对每个连续阶段做如下操作：

1. 通过神经网络馈送训练数据以产生输出。
2. 计算预期输出和预测输出之间的误差。
3. 基于该误差，计算神经网络权重和偏差的更新。
4. 将这些更新传播回网络。

提醒一下，具有一个隐藏层的网络实现过程如下所示（其中 `wHidden` 和 `wOut` 是隐藏层和输出层权重，`bHidden` 和 `bOut` 是隐藏层和输出层偏差）：

```
105     // Define the output of the neural network.
106     output := mat.NewDense(0, 0, nil)
107
108     // Loop over the number of epochs utilizing
109     // backpropagation to train our model.
110     for i := 0; i < nn.config.numEpochs; i++ {
111
112         // Complete the feed forward process.
113         hiddenLayerInput := mat.NewDense(0, 0, nil)
114         hiddenLayerInput.Mul(x, wHidden)
115         addBHidden := func(_, col int, v float64) float64 { return v +
bHidden.At(0, col) }
116         hiddenLayerInput.Apply(addBHidden, hiddenLayerInput)
117
118         hiddenLayerActivations := mat.NewDense(0, 0, nil)
119         applySigmoid := func(_, _ int, v float64) float64 { return
sigmoid(v) }
120         hiddenLayerActivations.Apply(applySigmoid, hiddenLayerInput)
121
122         outputLayerInput := mat.NewDense(0, 0, nil)
123         outputLayerInput.Mul(hiddenLayerActivations, wOut)
124         addBOut := func(_, col int, v float64) float64 { return v +
bOut.At(0, col) }
125         outputLayerInput.Apply(addBOut, outputLayerInput)
126         output.Apply(applySigmoid, outputLayerInput)
127
128         // Complete the backpropagation.
```

仅供非商业用途或交流学习使用

```
129         networkError := mat.NewDense(0, 0, nil)
130         networkError.Sub(y, output)
131
132         slopeOutputLayer := mat.NewDense(0, 0, nil)
133         applySigmoidPrime := func(_, _ int, v float64) float64 { return
sigmoidPrime(v) }
134         slopeOutputLayer.Apply(applySigmoidPrime, output)
135         slopeHiddenLayer := mat.NewDense(0, 0, nil)
136         slopeHiddenLayer.Apply(applySigmoidPrime,
hiddenLayerActivations)
137
138         dOutput := mat.NewDense(0, 0, nil)
139         dOutput.MulElem(networkError, slopeOutputLayer)
140         errorAtHiddenLayer := mat.NewDense(0, 0, nil)
141         errorAtHiddenLayer.Mul(dOutput, wOut.T())
142
143         dHiddenLayer := mat.NewDense(0, 0, nil)
144         dHiddenLayer.MulElem(errorAtHiddenLayer, slopeHiddenLayer)
145
146         // Adjust the parameters.
147         wOutAdj := mat.NewDense(0, 0, nil)
148         wOutAdj.Mul(hiddenLayerActivations.T(), dOutput)
149         wOutAdj.Scale(nn.config.learningRate, wOutAdj)
150         wOut.Add(wOut, wOutAdj)
151
152         bOutAdj, err := sumAlongAxis(0, dOutput)
153         if err != nil {
154             return err
155         }
156         bOutAdj.Scale(nn.config.learningRate, bOutAdj)
157         bOut.Add(bOut, bOutAdj)
158
159         wHiddenAdj := mat.NewDense(0, 0, nil)
160         wHiddenAdj.Mul(x.T(), dHiddenLayer)
161         wHiddenAdj.Scale(nn.config.learningRate, wHiddenAdj)
162         wHidden.Add(wHidden, wHiddenAdj)
163
164         bHiddenAdj, err := sumAlongAxis(0, dHiddenLayer)
165         if err != nil {
166             return err
167         }
168         bHiddenAdj.Scale(nn.config.learningRate, bHiddenAdj)
169         bHidden.Add(bHidden, bHiddenAdj)
170     }
```

将该实现详细列出，用以了解发生了什么。

产生输出的前馈过程执行了以下操作：

1. 将输入数据与隐藏层权重相乘，添加隐藏层偏差，并应用 sigmoid 激活函数来计算隐藏层 `hiddenLayerActivations`（前面代码段中的第 112 至 120 行）的输出。
2. 将输出层权重同 `hiddenLayerActivations` 相乘，然后添加输出层偏移，并应用 sigmoid 激活函数来计算 `output`（第 122 至 126 行）。

注意，在前馈过程中，从输入层的输入数据开始，通过隐藏层前进，直到得到

仅供非商业用途或交流学习使用

输出。

在前馈过程之后，需要计算权重和偏差的最优更新。正如在通过本附录的梯度下降部分后所预期的那样，梯度下降法非常适合找到这些权重和偏差。上面代码段中的第 129 至 144 行实现了 SGD。

最后，需要通过第 147 行到第 169 行的网络向后应用这些更新。这是更新的后向传播，这也是反向传播名称的来历。这个过程没有什么特别之处，只是执行以下操作：

1. 将计算的更新应用于输出权重和偏差（第 147 至 157 行）。
2. 将计算的更新应用于隐藏层权重和偏差（第 159 至 169 行）。

注意如何从输出开始，回到应用更改的输入。

可以在这里找到关于反向传播的详细讨论，包括数学证明：http://neuralnetwork-sanddeeplearning.com/chap2.html

反向传播在 1986 年的 David Rumelhart、Geoffrey Hinton 和 Ronald Williams 的论文之后开始广泛使用。尽管整个行业都在神经网络中使用该方法，但 Geoffrey Hinton 最近表示他对反向传播持怀疑态度，并建议需要努力寻找替代方案。

仅供非商业用途或交流学习使用

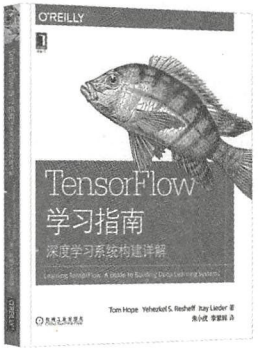
推荐阅读

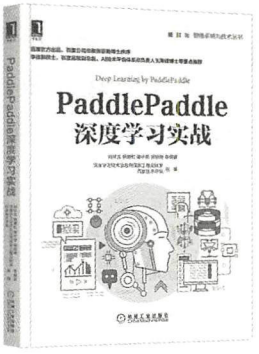
深度学习
核心技术、工具与案例解析

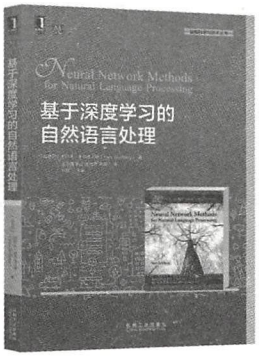
Deep Learning in Action
深度学习
主流框架和编程实战

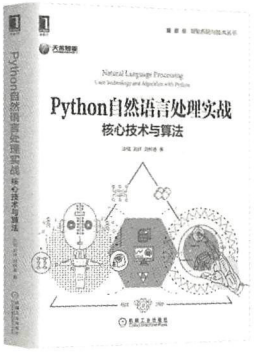
自然语言处理与深度学习
通过C语言模拟

深度学习
Java语言实现


O'REILLY
TensorFlow
学习指南
深度学习系统构建详解

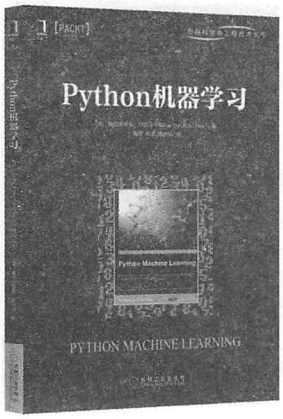

PaddlePaddle
深度学习实战

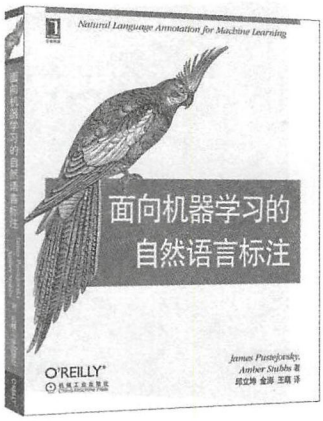

基于深度学习的
自然语言处理

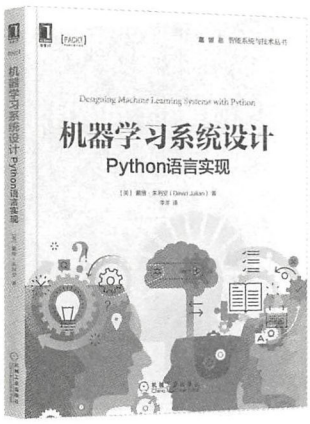

Python自然语言处理实战
核心技术与算法

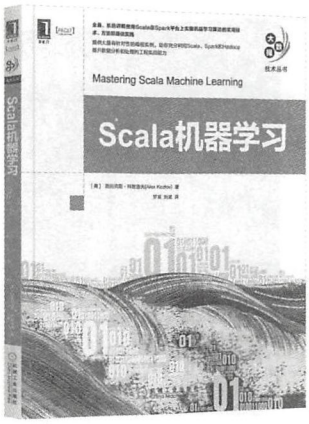
深度卷积网络
原理与实践

推荐阅读









Python机器学习

作者：Sebastian Raschka，Vahid Mirjalili ISBN：978-7-111-55880-4 定价：79.00元

机器学习：实用案例解析

作者：Drew Conway，John Myles White ISBN：978-7-111-41731-6 定价：69.00元

面向机器学习的自然语言标注

作者：James Pustejovsky，Amber Stubbs ISBN：978-7-111-55515-5 定价：79.00元

机器学习系统设计：Python语言实现

作者：David Julian ISBN：978-7-111-56945-9 定价：59.00元

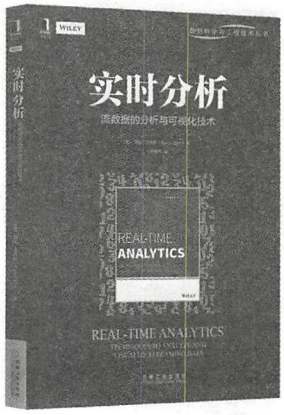
Scala机器学习

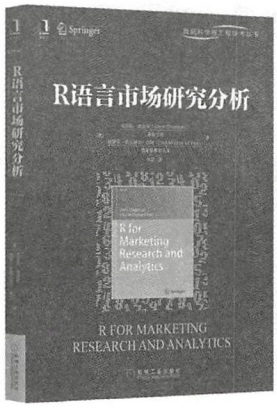
作者：Alexander Kozlov ISBN：978-7-111-57215-2 定价：59.00元

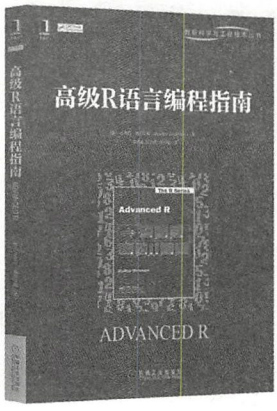
R语言机器学习：实用案例分析

作者：Dipanjan Sarkar，Raghav Bali ISBN：978-7-111-56590-1 定价：59.00元

推荐阅读







统计学习导论——基于R应用

作者：加雷斯·詹姆斯 等 ISBN：978-7-111-49771-4 定价：79.00元

应用预测建模

作者：马克斯·库恩 等 ISBN：978-7-111-53342-9 定价：99.00元

实时分析：流数据的分析与可视化技术

作者：拜伦·埃利斯 ISBN：978-7-111-53216-3 定价：79.00元

数据挖掘与商务分析：R语言

作者：约翰尼斯·莱道尔特 ISBN：978-7-111-54940-6 定价：69.00元

R语言市场研究分析

作者：克里斯·查普曼 等 ISBN：978-7-111-54990-1 定价：89.00元

高级R语言编程指南

作者：哈德利·威克汉姆 ISBN：978-7-111-54067-0 定价：79.00元

作者简介

丹尼尔·怀特纳克（Daniel Whitenack）是一位工作在Pachyderm(@pachydermIO)的资深数据科学家。Daniel开发出革命性的、分布式的数据流水线，其中包括预测模型、数据可视化、统计分析等。他在世界各大会议均有过演讲（GopherCon、JuliaCon、 PyCon、ODSC、 Spark Summit等）。目前在普渡大学(@LifeAtPurdue)教授数据科学/工程，同时也与Ardan实验室共同维护Jupyter项目的Go kernel。而且还积极为各种开源数据科学项目做出贡献。

译者简介

谢文江

毕业于北京工业大学计算机学院，曾服务于Motorola、IBM，目前就职于DELL EMC，担任质量主管工程师。

姜明魁

毕业于北京大学信息管理系，现任职全球500强知名车企，担任核心应用技术解决方案架构师。曾就职于全球顶级软件及云服务方案厂商，任云技术架构师。在虚拟化核心技术、云计算以及机器学习方面有丰富的经验和深入的研究。

机器学习

Go语言实现

MACHINE LEARNING WITH GO

本书旨在帮助你成为高效、创新的数据分析师，能使用Go语言构建健壮而有价值的应用。因此，本书不仅会清楚地介绍利用Go语言构建预测模型的技术，还会帮助你理解如何在真实场景中应用机器学习工作流程。

本书向你展示了如何在机器学习中提高效率，同时生产出保持高度完整性的应用，还给出一些模式，用以应对在将机器学习集成到工程组织中时经常遇到的挑战。

通过阅读本书，你将：

- 了解数据收集、组织、解析和清理
- 探索矩阵、线性代数、统计学和概率论
- 了解如何评估和验证模型
- 了解回归、分类和集群
- 了解神经网络和深度学习
- 利用时序模型和异常检测
- 掌握部署、分布分析和模型的技术
- 优化机器学习流程技术

[PACKT] PUBLISHING

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

上架指导：计算机/机器学习
ISBN 978-7-111-60979-7
9 787111 609797
定价：59.00元